



开发人员专业技术丛书

# MySQL 权威指南

(原书第2版)

MySQL: The definitive guide to using, programming,  
and administering MySQL 4 (Second Edition)

(美) Paul DuBois 著

杨涛 杨晓云 王群 等译

**SAMS**



机械工业出版社  
China Machine Press



# MySQL 权威指南 (原书第2版)

MySQL: The definitive guide to using, programming, and administering MySQL 4 (Second Edition)

作为开放源代码运动的产物之一, MySQL 关系数据库管理系统越来越受到人们的青睐, 应用范围也越来越广。闻名遐迩的速度和易用性使 MySQL 特别适用于 Web 站点或应用程序的数据库后端的开发工作。

如果你想高效且富有成果地使用 MySQL, 就应该好好读读本书。本书对以下几个方面进行了全面细致的讨论: 如何建立 MySQL 数据库、如何把 MySQL 与 PHP 或 Perl 结合起来以生成动态的 Web 网页、如何管理 MySQL 服务器。

## 作者介绍

**Paul DuBois** 是一位作家, 一名数据库管理员, 同时也是开放源代码和 MySQL 阵营里的一位旗手。他曾参与过 MySQL 在线文档的编写工作。除这本书以外, 他的主要著作还包括《MySQL and Perl for the Web》、《MySQL Cookbook》、《Using csh and tcsh》以及《Software Portability with imake》等。

“在我曾阅读过的各种技术论著中, 这本书是最好的技术书籍之一。”

——C & C++ 用户联合会, 《C Vu》杂志主编 Gregory Haley

“对于这本用户指南加参考手册形式的著作, 我只能用‘无出其右’来形容。我的结论是: 在 MySQL 数据库的日常使用和维护方面, 只要有了这本书就可以高枕无忧了。”

——《Web Techniques》杂志主编 Eugene Kim

封面设计 / 江丽萍

ISBN 7-111-13477-X



华章图书

网上购书: [www.china-pub.com](http://www.china-pub.com)

北京市西城区百万庄南街1号 100037

读者服务热线: (010)68995259, 68995264

读者服务信箱: [hzedu@hzbook.com](mailto:hzedu@hzbook.com)

<http://www.hzbook.com>

ISBN 7-111-13477-X/TP · 3328

定价: 98.00 元



开发人员专业技术丛书

# MySQL 权威指南

(原书第2版)

MySQL: The definitive guide to using, programming,  
and administering MySQL 4 (Second Edition)

(美) Paul DuBois 著

杨涛 杨晓云 王群 等译



机械工业出版社  
China Machine Press



MySQL 是基于SQL的客户/服务器模式的关系数据库管理系统,它具有功能强大、使用简单、管理方便、运行速度快、安全可靠性强等优点,特别适用于Web站点或应用程序的数据库后端的开发工作。另外,用户可利用许多语言编写访问MySQL数据库的程序。本书通过两个样板数据库,详细介绍了MySQL的基本概念、基本技巧、编程方法、管理特点以及第三方工具(如PHP和Perl)的使用方法。第2版保留了第1版的优点,同时增加了MySQL 4.0.1和相关编程语言接口的最新信息。

本书内容完善、条理清晰,适合数据库、网络开发与管理等人员参考。

Authorized translation from the English language edition entitled *MySQL: The definitive guide to using, programming, and administering MySQL 4* (ISBN: 0-7357-1212-3), 2e by Paul DuBois, published by Pearson Education, Inc., publishing as Sams Publishing, Copyright © 2003 by Sams Publishing.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanic, including photocopying, recording, or by any information storage retrieval system, without permission of Pearson Education, Inc.

Chinese simplified language edition published by China Machine Press.

Copyright © 2004 by China Machine Press.

本书中文简体字版由美国Pearson Education培生教育出版集团授权机械工业出版社独家出版。未经出版者书面许可,不得以任何方式复制或抄袭本书内容。

版权所有,侵权必究。

本书版权登记号:图字:01-2003-1295

图书在版编目(CIP)数据

MySQL权威指南(原书第2版)/(美)杜波依斯(DuBois, P.)著;杨涛等译.-北京:机械工业出版社,2004.1

(开发人员专业技术丛书)

书名原文:MySQL: The definitive guide to using, programming, and administering MySQL 4, Second Edition

ISBN 7-111-13477-X

I. M ... II. ①杜 ... ②杨 ... III. 关系数据库-数据库管理系统, MySQL IV. TP311.138

中国版本图书馆CIP数据核字(2003)第112672号

机械工业出版社(北京市西城区百万庄大街22号 邮政编码 100037)

责任编辑:迟振春

北京昌平奔腾印刷厂印刷·新华书店北京发行所发行

2004年1月第1版第1次印刷

787mm×1092mm 1/16·59.5印张

印数:0 001-4 000册

定价:98.00元

凡购本书,如有倒页、脱页、缺页,由本社发行部调换  
本社购书热线:(010) 68326294



## 前 言

无论是在商业、科研和教育等方面的传统应用项目里，还是作为因特网搜索引擎的后端支持，关系数据库管理系统（relational database management system, RDBMS）在许多场合都是一种极其重要的工具。如果没有充足的信息资源以及一整套良好的管理手段，建立一个良好的数据库系统就无从谈起，但有无充足的财力也是很多企事业单位能否建立起自己的数据库系统的关键因素之一。从历史情况看，数据库系统一直是一项昂贵的财产——无论是软件本身还是后续的技术支持，供货商从来都是漫天要价。此外，为了获得令人满意的性能表现，数据库引擎还往往会对计算机硬件有一些特殊的要求，而这又将使数据库系统的运营成本大大增加。

计算机硬件和软件在最近几年里的发展已经使这种情况得到了一定程度的改善。个人电脑的价格越来越低，性能却越来越高；而高性能的操作系统，如BSD UNIX操作系统的几种变体（例如FreeBSD、NetBSD、OpenBSD等）以及各种Linux版本（例如RedHat、Caldera、LinuxRPC等），也不断通过价格低廉的光盘被大量移植到个人电脑上，有些软件甚至可以通过因特网免费获得。

免费的操作系统大大增强了个人电脑的处理能力，与此同时，诸如GNU C编译器gcc之类的软件开发工具不仅在品种上越来越多，而且在品质和功能上也越来越完善。通过这些努力，高品质软件已经不再是普通用户可望而不可及的东西，只需付出很少的代价，任何人都能得到他想要的软件。这正是开放源代码（Open Source）运动的目的所在，而且它已经能够为我们提供多种重要的软件产品了，因特网上使用范围最广的Web服务器Apache就是一个最好的例子。“开放源代码”运动的成功范例还包括非常适合用来编写脚本程序的Perl语言、非常便于编写动态Web页面的PHP语言等等。与此形成鲜明对照的是，如果你决定采用某种专利性的商业化解决方案，就不得不忍受其供货商的漫天要价；而且，即使你为这个专利性的商业化解决方案支付了很高的费用，却极可能根本看不到它的源代码。

“开放源代码”运动也使数据库软件变得越来越容易获得。比如，PostgreSQL就是一种可以免费获得的数据库管理系统。而最近一个时期以来，有些商业化数据库管理系统（比如Informix和Oracle）的供货商也开始为Linux等操作系统免费提供它们的软件产品。但令人遗憾的是，这些免费提供的商业化数据库系统通常只包括二进制代码形式，而且不提供技术支持，这就使它们的用处和价值大打折扣。

MySQL是另一种低成本的数据库解决方案，它是一种基于SQL的客户/服务器模式的关系数据库管理系统，最初起源于欧洲的斯堪地那维亚半岛。MySQL由以下组件构成：一个SQL服务器、一些用来访问该服务器的客户端程序、一套用来对数据库进行管理的软件工具，以及一个为自行编写程序的数据库用户提供的程序设计接口。

MySQL起源于Michael Widenius（绰号Monty）在1979年为瑞士的TcX公司开发的UNIREG数据库工具。到了1994年，TcX公司开始寻求一种能够用来开发Web应用的SQL服务器。TcX公司对几种商业化的服务器进行了测试，但对它们在处理TcX公司的大数据表时的速度都不太满意。该



公司还测试了mSQL，但它又缺少某些必要的功能。因此，Monty开始开发一种新的服务器。因为mSQL有一些能够免费获得的软件工具，所以新服务器的程序设计接口被有意识地设计成与mSQL所使用的程序设计接口非常相似——采用相似的程序设计接口将大大减少把免费的软件工具移植到MySQL上的工作量。

到了1995年，Detron HB公司的David Axmark开始在因特网上推广和发行TcX公司研发的MySQL。David为MySQL编写了使用文档，并给MySQL增加了利用GNU组织的configure工具进行安装配置的功能。适用于Linux和Solaris系统的MySQL 3.11.1的二进制版本于1996年开始面世。如今，MySQL不仅能够在多种计算机平台上运行，而且还同时提供有二进制版本和源代码版本。MySQL的发行、技术支持与培训工作目前由专门为此组建的MySQL AB公司负责。

MySQL的开发和完善工作目前仍在不断地进行当中。发展到今天，大型数据库系统所必须具备的某些功能，比如事务处理、数据行锁定、外键支持以及数据镜像等，都已经被添加到MySQL里了，这使不少原来只考虑采购“大引擎”而对MySQL不屑一顾的用户开始重新审视和评估MySQL。

MySQL是一个“开放源代码”项目，在绝大多数场合都能免费使用，这使它在“开放源代码”界中享有很高的知名度。但必须指出的是，MySQL的知名度并不仅仅局限于“开放源代码”运动的支持者。的确，MySQL原本是运行在个人电脑上的（事实上，MySQL的很多开发工作都是在低廉的Linux系统上完成的），但它完全可以被移植并运行在商业化的操作系统（比如Solaris、Mac OS X、Windows）以及各种企业级的服务器硬件上。此外，MySQL的运行性能绝不逊色于任何一种你打算与之进行比较的数据库系统，即使面对拥有百万级数据记录的大型数据库，它也能游刃有余。

在功能强大但价格低廉的硬件设备上运行着免费的操作系统，越来越多的人在范围越来越广的硬件系统上拥有了越来越强大的计算能力，这是未来呈现在我们面前的一幅崭新画面，而MySQL在这幅画面里牢牢占据着一个显著的位置。获得强大计算能力的门槛正变得越来越低，而大型数据库解决方案对普通用户和中小企业来说也已经不再是可望而不可及的了。在过去，高性能的RDBMS只是广大中小企业的梦想，而现在，只需付出极低的成本和代价就能享用到这些东西。这一点对个人用户而言就更加突出了。就拿我本人来说吧，我有一台苹果电脑公司出品的iBook笔记本电脑，在它的Mac OS X操作系统上，我同时使用着MySQL以及Perl、Apache和PHP。这使我能够随时随地进行工作，而这一解决方案的总成本只是购买iBook笔记本电脑的价钱而已。

## 为什么选用MySQL

如果你正在寻求一种完全免费或者价格比较低廉的数据库管理系统，那就不妨对以MySQL、PostgreSQL（一种完全免费但供货商不提供技术支持的数据库引擎）为代表的有关软件进行一下对比评估。在对MySQL和其他数据库系统进行评估之前，首先要把对自己最为重要的因素弄清楚，需要从运行性能、技术支持、特色功能（比如与SQL的兼容程度和可扩展性等）、许可条款、购买价格等多方面进行全面考虑。依我之见，MySQL在以下方面有比较吸引人的优势：

- 运行速度——MySQL的运行速度相当快。MySQL的开发人员认为它是目前最快的数据库管



理系统。可以从MySQL的Web站点上的性能对照页面中查到有关数据：<http://www.mysql.com/benchmark.html>。

- 易使用性——MySQL是一种简单易用的高性能数据库管理系统，与其他大型数据库管理系统相比，MySQL的安装和管理工作要容易得多。
- 查询语言支持——MySQL支持SQL语言，该语言是所有现代数据库系统的首选查询语言。
- 功能丰富——MySQL允许多个客户端同时与服务器建立连接。客户端可以同时打开并使用多个数据库。可以通过好几种办法（比如命令行客户程序、Web浏览器、X窗口系统下的客户程序等等）对MySQL数据库进行交互式访问，在输入查询命令后可立刻看到查询结果。此外，MySQL还提供了C、Perl、Java、PHP、Python等多种程序设计语言的编程接口。你还可以通过支持ODBC（Open Database Connectivity，开放数据库连接，一种由微软公司开发的数据库通信协议）功能的应用程序来访问MySQL数据库。也就是说，你既可以选用现成的客户程序来访问MySQL数据库，也可以根据具体的应用情况来编写相关的软件。
- 优异的联网和安全性能——MySQL是网络化的数据库系统，用户可以从因特网上的任意地点访问MySQL数据库，完全可以把数据与世界上任何地方的任何人共享。同时，MySQL还具备完善的访问控制机制，这就使那些不应该看到你数据的人不能看到你的数据。此外，为了提供更进一步的安全措施，MySQL还支持使用SSL（Secure Socket Layer，安全套接字层）协议的加密连接。
- 可移植性——MySQL既能够运行在多种版本的UNIX操作系统上，也能够运行在诸如Windows或者OS/2之类的非UNIX系统上。MySQL可以运行在从家用电脑到企业级高端服务器的全系列硬件设备上。
- 短小精悍——与某些商业化数据库系统巨大的硬盘空间消耗量相比，MySQL发行版本的硬盘占用量相对小得多。
- 成本低廉——MySQL是一个“开放源代码”项目，只要你遵守GNU组织的通用公共许可证（General Public License，GPL）条款，就可以不受任何限制地使用。这意味着MySQL在大多数情况下都是免费的（但如果你想通过销售MySQL或者提供与之相关的服务来赚钱，那就必须得到MySQL AB公司的许可和授权）。
- 来源广泛——MySQL很容易获得，只要你有Web浏览器，就能从许多地方下载到它。如果你想知道其中某个组件的工作原理或者对它的某个算法感到好奇，完全可以通过它的源代码来进行钻研。如果你不喜欢它的某个组件，也完全可以自行加以修改。如果你认为自己发现了一个bug（程序漏洞），也完全可以报告给有关开发人员。

MySQL的技术支持怎么样？这个问题问得好——没有必要的技术支持，很难用好一种数据库。我本人当然希望这本书能够满足你在数据库方面的全部需要，但在实际工作中，你肯定会遇到一些我根本没有想到或者没有在这本书里讨论到的问题。不过你用不着为这件事担心——网上有大量与MySQL有关的资源，足以满足你的任何要求。MySQL是免费的，但这并不意味着你在安装和使用MySQL时是孤军奋战。下面是一些最方便的MySQL资源：

- MySQL的发行版本都带有MySQL Reference Manual（MySQL参考手册），也可以在网上找到它。MySQL用户对这本手册给予了很高的评价，这一点非常重要——如果没有一本好用



的参考手册，再好的软件产品也会贬值。

- 如果你想得到正规的培训或者专业化的技术支持，可以报名参加MySQL AB公司开设的培训课程或者与该公司签订技术支持合同。
- MySQL社团有一个非常活跃的邮件列表，任何人都能订阅。这个邮件列表有很多专家级的参与者，许多MySQL的开发人员都是它的常客。作为提供技术支持的电子资源，很多订阅者都认为它物有所值。

MySQL社团（包括开发人员和普通用户在内）是一个非常团结互助的群体。贴在邮件列表上的求助帖子经常会在几分钟内得到回复。如果有人报告说发现了一个bug（程序漏洞）并得到确认，开发人员就会马上发布一个修复方案并经由因特网迅速传遍整个社团。如此之快的响应速度和如此认真负责的态度在这一领域里的其他公司的技术支持服务站点上是看不到的。（你体会过其他公司的技术支持服务吗？我可是吃过不少苦头的。在陷入麻烦的时候，谁都希望能够尽快得到专业的技术支持。如果一种办法是苦等供货商在他们自认为适当的时候才来“帮助你”，而另一种做法是你发一个求助帖子并在自己认为适当的时候去看看有没有回音，你会选择哪一种办法？）

如果你正打算挑选一种数据库产品，那么MySQL绝对是一个值得考虑的候选者。试用MySQL既无风险，也不需要花费金钱。如果你在评估过程中遇到了问题，还可以通过我刚才介绍的邮件列表寻求帮助。当然了，进行这样的评估必定会花费一些时间，但无论你原来计划使用哪一种数据库产品，都要花些时间去进行评估，所以我想这一点应该不会成为你拒绝给MySQL一个机会的理由——在我看来，与很多其他的数据库产品相比，你花在安装和测试MySQL上的时间要更加物有所值。

## 如果已经运行着其他RDBMS该怎么办

如果你现在已经在使用某种数据库管理系统，是否应该转换到MySQL上去呢？我的回答是：没有这个必要。如果你对现有的系统很满意，为什么还要给自己找麻烦呢？要是你对自己正在使用的数据库产品有所不满，那就绝对应该给MySQL一个机会。也许你觉得自己现有系统的性能不太好，也许它是一个专利性的产品而你又不想吊死在一棵树上，也许你想更换现有的硬件设备，可现有的软件系统却不支持，也许你现有的软件都是二进制代码而你更希望得到一种能够提供源代码的系统，也许你只是嫌它花钱太多……这一切都是你应该给MySQL一个机会的理由。你可以先通过这本书来熟悉一下MySQL的功能，再到MySQL邮件列表上提几个问题，然后再根据各方面的具体情况慎重地做出抉择。

如果你正准备从另外一种SQL数据库转移到MySQL上，建议你先到MySQL的Web站点上的产品对比页面（<http://www.mysql.com/information/crash-me.php>）上去看一看，然后再到本书的有关章节里去熟悉一下MySQL所支持的数据类型和有关的SQL概念。如果你现有的RDBMS系统所支持的SQL版本与MySQL所支持的SQL版本之间的差异比较大，那么现有应用程序的移植工作就可能要花费相当大的努力和代价才能成功——如果真是这样，请三思而后行。

不过，就算你现有的数据库系统“古老”到连SQL也“听”不懂的地步，也并不意味着你只



能望洋兴叹。不妨从先移植几个有代表性的数据表来开始评估工作，你也许会发现实际情况并没有想像的那么困难。我本人就曾有过这样一次经历：我受命把一个根本不“懂”SQL的RDBMS系统移植到MySQL上去，可这两种系统的编程语言不仅没有一点儿可供利用的近似之处，而且老系统里的某些数据类型甚至连对应的SQL语言元素都不存在。这个项目不仅需要对网络访问方法进行转换，还需要修改很多基于屏幕的数据录入程序和已经定制成套的查询语句。这个项目让我忙活了大约一个半月的时间，但最终的效果却相当不错。

## 随MySQL提供的软件工具

MySQL的发行版本都附带有以下几种工具程序：

- MySQL服务器——运转整个MySQL数据库系统的引擎，对MySQL数据库的访问和操作都要通过它才能实现。
- 用来访问MySQL服务器的客户端程序——其中包括一个供用户直接提交查询并查看其结果的交互式程序和几个用来对数据库站点进行管理和维护的工具程序。有一个工具程序是用来调控MySQL服务器的，另几个工具程序则负责完成有关数据的导入导出和用户访问权限的检查工作。
- 供用户自行开发应用程序的客户程序开发库——这个开发库是用C语言写的，所以可以用C语言来编写自己的客户端程序。此外，这个开发库还提供了一些供其他程序设计语言使用的第三方接口。

除MySQL本身所提供的软件程序外，还可以通过因特网等多种途径找到很多高质量的第三方工具软件。有很多极富聪明才智的人也使用着MySQL，他们喜欢自己编写一些小程序来提高工作效率，也喜欢把自己的成果拿出来与大家分享。在这些第三方工具软件里，有些能帮助你更加得心应手地使用MySQL，有些能把MySQL的功能进一步扩展到建设Web站点。

## 本书能让你学到哪些东西

本书的目的是帮助大家高效率地掌握MySQL的使用方法，从而高效率地完成自己的本职工作。从中将会学到怎样才能有效地把信息资料录入数据库、怎样才能构造出优化的查询语句以迅速获得有关问题的答案数据。

学习和使用SQL并不要求你已经是一名程序员。本书的内容重点之一就是向大家介绍SQL的工作原理。但熟悉SQL的语法并不代表你掌握了SQL的使用技巧。而本书的另一个重点就是介绍MySQL的独家功能以及它们的实际用法。

大家还将学到怎样才能把MySQL与其他软件工具结合起来使用。本书还将向大家介绍如何通过MySQL与Perl或PHP语言来为数据库的查询结果生成动态Web页面。自行编写MySQL数据库访问程序也将是本书的学习重点之一。自行编写的这些程序应该最符合你应用项目的具体要求，从而大大拓展MySQL的功能，增加它的应用价值。

对于那些负责MySQL数据库系统的安装和管理工作的人员，这本书将提供有关职责及其具体实现方式。你将学会如何建立用户账户、如何对数据库进行备份以及如何保证数据库的安全。



## 本书各章内容介绍

本书分为四大部分。

### 第一部分：MySQL基础知识

- 第1章，MySQL和SQL入门——主要内容包括：MySQL的用途与用法；交互式MySQL客户程序的使用方法；SQL基础知识；MySQL的常用功能。
- 第2章，MySQL数据库里的数据——主要内容包括：MySQL为描述数据而提供的数据列类型（column type）；各种类型的特点和局限性；各种类型的使用时机和使用方法；如何在类似的数据列类型中做出选择；表达式的求值办法；各类型之间的转换机制。
- 第3章，MySQL SQL语法及其使用——如今，各种主流的RDBMS都能识别和理解SQL语言，但各种数据库引擎所使用的SQL语言彼此有着细微的差异。本章的介绍重点是使MySQL有别于其他数据库系统的特色功能。
- 第4章，查询优化——如何使查询运行得更有效率。

### 第二部分：MySQL程序设计接口

- 第5章，MySQL程序设计简介——主要内容包括：MySQL提供的几种应用程序设计接口（application programming interface, API）以及本书所涉及的几种API之间的详细对比。
- 第6章，MySQL应用程序设计接口：C语言——如何利用MySQL发行版本自带的客户程序开发库所提供的API来编写C语言程序。
- 第7章，MySQL应用程序设计接口：Perl DBI——如何利用DBI模块来编写Perl脚本，包括用于独立单机上的普通脚本和用于Web站点上的CGI脚本。
- 第8章，MySQL应用程序设计接口：PHP语言——如何利用PHP脚本语言来编写用来访问MySQL数据库的动态Web页面。

### 第三部分：MySQL系统管理

- 第9章，MySQL系统管理简介——主要内容包括：数据库管理员的工作职责；怎样才能让数据库站点运行得更成功。
- 第10章，MySQL的数据目录——详细介绍MySQL数据目录（即MySQL用来存放各种数据库文件和辅助性文件的区域）的组织布局和内容。
- 第11章，MySQL数据库系统的日常管理——主要内容包括：如何正确地完成数据库系统的开机和关机任务；如何在MySQL系统里建立用户账户；如何对各种日志文件进行维护和管理；如何对InnoDB表空间进行配置；如何对数据库服务器进行优化；如何运行多个服务器；如何建立镜像服务器。
- 第12章，MySQL安全技术——主要内容包括：如何提高MySQL系统的安全水平以抵御各种入侵和破坏，入侵可能来自数据库服务器主机，也可能来自经网络而连接的其他客户端；如何配置MySQL服务器以支持SSL上的安全连接。
- 第13章，MySQL数据库的备份、维护和修复——主要内容包括：如何通过预防性措施来降低灾难的发生几率；如何对数据库进行备份；如何在灾难真的发生时（要知道，再好的预防性措施也不意味着灾难永远都不会发生）尽快恢复系统的运转。



#### 第四部分：附录

- 附录A，获得并安装有关软件——如何获得并安装本书所提到的有关软件和常用工具程序。
- 附录B，数据列类型指南——MySQL数据列类型的详细描述。
- 附录C，操作符与函数用法指南——在SQL语句中用来编写表达式的操作符和函数的详细描述。
- 附录D，SQL语法指南——MySQL所支持的各种SQL语句的详细描述。
- 附录E，MySQL程序用法指南——MySQL发行版本所提供的有关程序的详细描述。
- 附录F，C API指南——MySQL C语言客户程序开发库所提供的数据类型和函数功能的详细描述。
- 附录G，Perl DBI API指南——Perl DBI模块提供的方法和属性的详细描述。
- 附录H，PHP API指南——PHP为支持MySQL而提供的函数功能的详细描述。
- 附录I，挑选ISP——作为顾客，怎样在提供MySQL访问服务的ISP（Internet Service Provider，因特网服务提供商）中做出选择；作为ISP，怎样才能为顾客提供更好的MySQL接入服务。

#### 如何阅读本书

在阅读和学习本书的时候，希望大家能亲自对各章中的示例进行尝试。如果你的计算机上还没有安装MySQL，请先亲自或请别人来把它安装好，再把样板数据库sampdb的有关文件（本书的许多示例都要用到这个数据库）安装到你的机器里。获得和安装有关组件的办法与步骤可以在附录A里查到。

如果你是一位MySQL数据库系统或SQL语言的新手，请从本书的第1章开始学习，第1章所介绍的MySQL与SQL的基本概念和使用入门对加快本书后续章节的学习进度有很大的帮助。然后再前进到第2章和第3章去学习如何描述和使用自己的数据，这样，就能有针对性地根据自己的具体应用项目去探索各种MySQL功能了。

即使你已经具备了一些SQL方面的经验，也应该从第2章和第3章开始入手。不同的RDBMS系统所实现的SQL功能彼此有着细微的差别，大家应该首先把MySQL有别于自己所熟悉的其他RDBMS系统的那些东西弄清楚。

如果你已经有了一些MySQL方面的经验但还需要进一步了解某些特定工作的原理和细节，请把这本书当做一本参考大全，并根据自己的具体需要有选择地查阅本书的有关章节。从把这本书当做一本参考大全的角度看，书后的各种附录有着非常高的参考价值。

如果你的兴趣在于怎样才能编写出具备MySQL数据库访问功能的程序，请从第5章开始去学习有关各种API的章节。如果你想为自己的数据库开发一些便于使用的基于Web的前端访问程序，或者想为自己的数据库Web站点开发一些后端程序以便给自己的Web站点增添动态内容，请阅读第7章和第8章的内容。

如果你打算对MySQL和自己正在使用的RDBMS进行对照评估，也可以在本书里找到答案。如果你想了解MySQL与现有的SQL系统有何异同，请阅读本书第一部分中专门讨论数据类型和SQL语法的有关章节；如果你打算自行开发应用程序，请阅读本书第二部分中讨论有关应用程序

设计接口的章节；如果你想了解MySQL提供有何种级别的数据库管理功能，请阅读本书第三部分中的有关章节。如果你此前尚未使用过任何一种数据库，但正在对MySQL和其他数据库系统进行对比评估，这些内容也将有很大的帮助。

如果你想访问MySQL数据库并正为此而寻找着ISP，可以在附录I里查到一些挑选准则。这个附录还对希望提供MySQL服务以吸引新顾客或者希望进一步改善现有顾客服务水平的ISP们提供了一些忠告。

## 书中涉及的软件及其版本

在编写本书的时候，MySQL的最新版本是4.0系列，而4.1版本系列正在紧锣密鼓地开发和完善当中。因此，本书的讨论将主要集中在这两个版本系列上，但同时也会兼顾到较早出现的3.22和3.23系列版本中的有关功能。

至于本书所涉及的其他的软件包，它们的最新版本都足以满足书中示例的要求。下面是部分重要软件包的当前版本：

软 件 包	版 本
Perl DBI	1.32
Perl MySQL DBI驱动程序	2.1020
PHP	4.2.3
Apache	1.3.27 / 2.0.43
CGI.pm	2.87

本书提到的软件都可以在因特网上找到。附录A对获得并在系统上安装MySQL、Perl DBI、PHP、Apache、CGI.pm等软件及步骤进行了介绍。这个附录还给出了如何在系统上生成本书所使用的样板数据库以及本书论述程序设计问题的有关章节将会用到的示例程序的办法。

## 书中使用的符号体例

本书使用的符号体例如下：

主机名、文件名、目录名、命令、选项和Web站点都用普通英文字体表示。命令中需要由读者输入的部分用粗体英文字体表示。命令中的斜体字部分表示需要由读者把它替换为自己选择的内容。

此外，不同的命令行提示符代表了不同的命令执行方式。百分号（%）是最常见的命令行提示符，它表示有关命令可以在UNIX系统的shell提示符或者Windows系统的DOS提示符下使用。井号（#）提示符的意义比较特殊，它表示有关命令只能由UNIX系统的根用户（即root用户）执行；而“C:\>”提示符则表示有关命令是Windows下的专用命令。而在使用mysql程序的过程中发出的SQL语句都跟在提示符“mysql>”后面。

在SQL语句里，SQL关键字和函数名都用大写的英文字母写出，而数据库、数据表、数据列的名称则全部用小写字母写出。在语法描述中，方括号（[]）表示有关内容可选。

书中出现的“基于Windows NT的系统”代表由Windows NT衍生出来的各种Windows变体，包括Windows NT、Windows 2000和Windows XP，而Windows 95、Windows 98和Windows Me不包括在内。



## 其他资源

就个人而言，我希望本书能把大家想知道的关于MySQL的知识都收录在内。可万一你的问题没能在这本书里找到答案，又该怎么办呢？

以下是一些有关软件的Web站点：

软件包	主要的Web站点
MySQL	<a href="http://www.mysql.com/documentation/">http://www.mysql.com/documentation/</a>
Perl DBI	<a href="http://dbi.perl.org/">http://dbi.perl.org/</a>
PHP	<a href="http://www.php.net/">http://www.php.net/</a>
Apache	<a href="http://www.apache.org/">http://www.apache.org/</a>
CGI.pm	<a href="http://stein.cshl.org/WWW/software/CGI/">http://stein.cshl.org/WWW/software/CGI/</a>

这些Web站点上还有指向其他各种有用的信息资源（比如参考手册、常见问题解答（frequently asked-question, FAQ）文档以及各种邮件列表）的链接：

- 参考手册——参考手册是MySQL发行版本中自带的主要文档。这些文档的格式有很多种，网上还有它们的在线版本。PHP的使用手册也有好几种格式。DBI模块及其MySQL专用驱动程序的文档是彼此分开的：DBI的文档侧重于基本概念，而其MySQL专用驱动程序的文档则侧重于与MySQL有关的各种具体功能。
- FAQ文档——DBI、PHP、Apache各有各的FAQ文档。
- 邮件列表——本书所涉及的一些软件有它们各自的邮件列表。如果你打算使用并想用好某个工具软件，那最好去订阅一份与之有关的邮件列表。邮件列表上的论文集或文章汇总也很值得下载来仔细研究：如果你不熟悉某个软件工具的用法，你的很多问题就可能是很多人已经问过（并得到回答）无数次的了，你不必再提出类似的问题，因为它们的答案几乎都能在邮件列表的论文集或文章汇总里找到。

不同的邮件列表有不同的订阅办法，但你肯定能在下面这些URL地址处找到一些线索：

软件包	邮件列表订阅办法
MySQL	<a href="http://www.mysql.com/documentation/">http://www.mysql.com/documentation/</a>
Perl DBI	<a href="http://dbi.perl.org/">http://dbi.perl.org/</a>
PHP	<a href="http://www.php.net/support.php">http://www.php.net/support.php</a>
Apache	<a href="http://www.apache.org/foundation/maillinglists.html">http://www.apache.org/foundation/maillinglists.html</a>

- 其他Web站点——除官方Web站点外，书中涉及的某些软件工具还另有一些提供其他信息（如样板程序的源代码或者热门文章等）的站点。这些站点大都可以通过各有关官方站点上的链接找到。

### 使用在线MySQL参考手册

请定期查阅在线MySQL参考手册以获取最新的MySQL信息。该手册是与MySQL软件同步修改和更新的。

mysql 6.5.17

## 作者简介

Paul DuBois是一位作家，一名数据库管理员，同时也是开放源代码阵营里的一位旗手。除这本书以外，他的主要著作还包括*MySQL and Perl for the Web*（用MySQL和Perl开发Web应用）、*MySQL Cookbook*（MySQL使用进阶）、*Using csh and tcsh*（csh和tcsh的使用）以及*Software Portability with imake*（软件移植工具imake）等。

## 技术顾问简介

本书的出版离不开有关责任编辑人员的辛勤劳动，他们使这本书的内容更加充实和完备。这些细心的专业人士对书中的每一个技术细节都进行了核实和校对，正是因为他们的努力，才使本书满足广大读者对一本高质量的技术类书籍的期望和要求。

Shane Kirk毕业于肯塔基大学计算机科学系并获学士学位。他现在居住于美国俄亥俄州的辛辛那提市，是Opinion One公司（[www.opinionone.com](http://www.opinionone.com)）的数据库管理员和软件开发人员，Opinion One公司的业务重点是开发市场分析软件及有关的解决方案。

Hang T. Lau博士是位于加拿大蒙特利尔市的康考迪亚大学计算机科学系的一位助理教授。作为一名系统科学家，他有着超过20年的工作经验，他的主要研究方向包括电信网络规划、电信中的语音识别技术、无线移动网络系统等等。

## 致谢

感谢以下人员对本书第1版和第2版所给予的支持和帮助。

### 第1版

感谢以下人员对本书的校对、批评和指正：David Axmark、Vijay Chaugule、Chad Cunningham、Bill Gerrard、Jijo George John、Fred Read、Egon Schmid和Jani Tolonen。我还要特别感谢MySQL的主要作者——人称Monty的Michael Widenius，他不仅细心地校对了本书的手稿，而且还耐心地解答了我在本书写作过程中向他提出的数以百计的问题。如果读者还在这本书里发现了错漏，就只能怪我本人学艺不精了。我还要感谢Thomas Karlsson、Colin McKinnon、Sasha Pachev、Eric Savage、Derick H. Siddoway以及Bob Worthy，他们对这本书的初稿进行了认真的校对，并帮助我把这本书改进成现在的样子。

我还要衷心感谢New Riders出版社的有关工作人员，没有他们的支持与帮助，就不可能有这本书的面世。Laurie Petrycki是本书的主编。Katie Purdum是本书的责任编辑，她一直关心着本书的写作，并不断督促我抓紧时间完成这项工作。负责本书整理和文字工作的Leah Williams为这本书加了不少的班——尤其是在本书接近完成的最后阶段。本书的索引是由Cheryl Lenser和Tim Wright完成的。本书的项目编辑John Rahm也为这本书倾注了大量的心血。在此谨向以上人员表示我衷心的感谢。

最后，我还要感谢我的妻子Karen。为了支持我的工作，她推迟了她自己的一本书的写作和出版计划。正是有了她的忍耐和理解，我才能整天埋头于写作当中。没有她的支持，这本书就不



能如此顺利地得以完成，可以说，这本书里的每一页都有她的贡献。

## 第2版

技术顾问在本书的第2版里依然起了重要作用，他们对书中的错漏进行了认真的查找和纠正，对含混不清的地方进行了澄清。Hang Lau和Shane Kirk是本版的技术顾问。我还要感谢Monty Widenius、Alexander Barkov、Jani Tolonen等MySQL开发人员，他们不厌其烦地解答了我的许多疑问，使这本书的内容得到了进一步的充实和完善。

感谢New Riders出版社的以下工作人员：出版人Stephanie Wall、助理编辑Chris Zahn、高级主编Lori Lyons、文字编辑Pat Kinyon、索引编辑Cheryl Lenser、排版人员Stacey RichwineDeRome。

最后，我还要感谢这本书的幕后英雄和第一读者——我的妻子Karen。没有她，这本书是不可能达到现在这个水平的。

## 请把你的想法告诉我们

作为本书的读者，你的意见和看法是最为重要的。希望大家能够不吝赐教，告诉我们这本书哪些地方做得不错，哪些地方还需要改进，哪些东西是你们想知道而我们又没有收录在其中的——总之，只要是读者的声音，我们都将认真听取。

大家可以通过电子邮件或普通信件直接与我联系，好让我了解你们对这本书的看法以及改进建议。但我要告诉大家的是，对于大家在学习本书时遇到的技术性问题，我不一定能帮得上忙；而且，因为我每天都会收到大量的电子邮件，所以可能无法对每一封邮件做出回复。

在来信的时候，请大家把本书的书名、作者以及你的姓名、电话或者电子邮件地址写清楚。我将认真对待各位读者的来信并与本书的其他作者和有关编辑人员进行交流。下面是我们的联系办法：

电子邮件：[devlib@sampublishing.com](mailto:devlib@sampublishing.com)

普通信件：Mark Taber

Associate Publisher

Developer's Library

Sams Publishing

201 West 103rd Street

Indianapolis, IN 46290 USA

---

除封面署名外，参加本书翻译的人员还有：王建桥、高文雅、胡建平、李艳、许玉新、李春卉、张玉亭、韩兰、杨楨和、王玉敏、郭静、许森、郝彬、孙晓红、亢海宏、王贵新、王冲、李京山、韩东升、张玉乔、郭连义、郭明等。

# 目 录

## 前言

## 第一部分 MySQL基础知识

第1章 MySQL和SQL入门 .....	2
1.1 MySQL概述 .....	2
1.2 样板数据库 .....	5
1.2.1 美国历史研究会 .....	6
1.2.2 考试记分项目 .....	8
1.2.3 关于样板数据库的说明 .....	8
1.3 数据库基本术语 .....	9
1.3.1 数据库的组织结构术语 .....	9
1.3.2 数据库查询语言术语 .....	11
1.3.3 MySQL的体系结构术语 .....	11
1.4 MySQL教程 .....	13
1.4.1 获得样板数据库 .....	13
1.4.2 最低配置要求 .....	14
1.4.3 建立和断开与服务器的连接 .....	15
1.4.4 发出查询命令 .....	17
1.4.5 创建数据库 .....	19
1.4.6 创建数据表 .....	20
1.4.7 添加新记录 .....	35
1.4.8 对信息进行检索 .....	39
1.4.9 删除或更新现有的数据记录 .....	70
1.5 交互式客户程序mysql的使用技巧 .....	72
1.5.1 简化连接过程 .....	73
1.5.2 减少查询命令的输入 .....	75
1.5.3 改变mysql客户程序的提示符 .....	78
1.6 今后各章的学习计划 .....	79
第2章 MySQL数据库里的数据 .....	80
2.1 MySQL的数据类型 .....	81
2.1.1 数值 .....	81

2.1.2 字符串(字符)值 .....	82
2.1.3 日期和时间值 .....	83
2.1.4 NULL值 .....	84
2.2 MySQL的数据列类型 .....	84
2.2.1 数据列类型概述 .....	84
2.2.2 数据表的创建 .....	86
2.2.3 数值类数据列类型 .....	87
2.2.4 字符串类数据列类型 .....	95
2.2.5 日期和时间类数据列类型 .....	108
2.3 序列与编号 .....	115
2.3.1 ISAM数据表里的AUTO_INCREMENT 数据列 .....	115
2.3.2 MyISAM数据表里的AUTO_INCREMENT 数据列 .....	117
2.3.3 HEAP数据表里的AUTO_INCREMENT 数据列 .....	119
2.3.4 BDB数据表里的AUTO_INCREMENT 数据列 .....	119
2.3.5 InnoDB数据表里的AUTO_INCREMENT 数据列 .....	119
2.3.6 使用AUTO_INCREMENT机制时的 注意事项 .....	119
2.3.7 强制MySQL不要复用已经用过的 序列值 .....	120
2.3.8 给数据表增加一个序列编号数据列 .....	121
2.3.9 重新编排现有的序列编号 .....	121
2.3.10 在不使用AUTO_INCREMENT机制的 情况下生成序列编号 .....	122
2.4 MySQL对字符集的支持 .....	124
2.4.1 MySQL 4.1之前版本对字符集 的支持 .....	125



2.4.2 MySQL 4.1及以后版本对字符集的支持 .....	126
2.5 选择数据列类型 .....	128
2.5.1 这个数据列将用来存放哪一种数据 .....	130
2.5.2 数据值是否都位于某个区间范围内 .....	132
2.5.3 有没有性能和效率方面的问题 .....	134
2.5.4 打算如何对有关数据进行比较 .....	136
2.5.5 是否要在某个数据列上建立索引 .....	136
2.5.6 数据列类型选择问题的内在联系 .....	137
2.6 表达式求值与类型转换 .....	137
2.6.1 书写表达式 .....	138
2.6.2 类型转换 .....	145
第3章 MySQL SQL语法及其使用 .....	153
3.1 MySQL的命名规则 .....	154
3.1.1 数据库组成元素的命名规则 .....	155
3.1.2 SQL语句对字母大小写的要求 .....	157
3.2 数据库的选定、创建、丢弃和变更 .....	158
3.2.1 数据库的选定 .....	158
3.2.2 数据库的创建 .....	158
3.2.3 数据库的丢弃 .....	159
3.2.4 数据库的变更 .....	159
3.3 数据表的创建、丢弃、索引和变更 .....	159
3.3.1 数据表类型 .....	159
3.3.2 数据表的创建 .....	164
3.3.3 数据表的丢弃 .....	171
3.3.4 数据表的索引 .....	171
3.3.5 变更数据表的结构 .....	176
3.4 获得关于数据库和数据表的信息 .....	180
3.4.1 确定MySQL服务器所支持的数据表类型 .....	181
3.4.2 检查数据表是否存在及其类型 .....	182
3.5 涉及多个数据表的查询操作 .....	183
3.5.1 单关联 .....	184
3.5.2 全关联 .....	184
3.5.3 左关联和右关联 .....	186
3.5.4 使用子选择 .....	190
3.5.5 涉及多个数据表的UNION查询 .....	193
3.6 涉及多个数据表的删除和修改操作 .....	198
3.7 事务处理 .....	199
3.7.1 事务处理机制的用途 .....	200
3.7.2 事务问题的非事务实现办法 .....	201
3.7.3 利用事务处理机制来保证语句的安全执行 .....	203
3.8 外键与引用完整性 .....	207
3.9 使用FULLTEXT全文本搜索 .....	213
3.10 代码注释 .....	218
3.11 MySQL不支持的特征 .....	219
第4章 查询优化 .....	221
4.1 索引的使用 .....	221
4.1.1 索引的优点 .....	222
4.1.2 索引的缺点 .....	224
4.1.3 挑选索引 .....	225
4.2 MySQL的查询优化程序 .....	227
4.2.1 查询优化程序的工作原理 .....	228
4.2.2 抑制优化程序给出的方案 .....	231
4.3 数据列类型与查询效率 .....	232
4.4 更有效地加载数据 .....	235
4.5 调度和锁定问题 .....	238
4.6 系统管理员所完成的优化 .....	241
4.6.1 查询缓存区 .....	243
4.6.2 与硬件有关的优化问题 .....	245
第二部分 MySQL程序设计接口	
第5章 MySQL程序设计简介 .....	248
5.1 为什么要自行编写MySQL程序 .....	248
5.2 可用于MySQL的API .....	251
5.2.1 C API .....	253
5.2.2 Perl DBI API .....	254
5.2.3 PHP API .....	255
5.3 选择API .....	257
5.3.1 运行环境 .....	257
5.3.2 性能 .....	258
5.3.3 开发周期 .....	260

5.3.4 可移植性 .....	262	7.1 Perl语言脚本程序的特点 .....	324
第6章 MySQL应用程序设计接口:		7.2 Perl DBI概述 .....	325
C语言 .....	263	7.2.1 DBI数据类型 .....	325
6.1 客户程序的制作流程 .....	264	7.2.2 一个简单的DBI脚本 .....	326
6.1.1 对系统的基本要求 .....	264	7.2.3 出错处理 .....	331
6.1.2 MySQL客户程序的编译和链接 .....	264	7.2.4 处理没有结果集的查询 .....	334
6.2 客户程序1——连接到服务器 .....	266	7.2.5 处理有结果集的查询 .....	335
6.3 客户程序2——增加出错检查功能 .....	269	7.2.6 引号问题 .....	345
6.4 客户程序3——运行时获取连接参数 .....	273	7.2.7 占位符与参数绑定 .....	348
6.4.1 访问选项文件的内容 .....	274	7.2.8 把查询结果绑定给脚本变量 .....	350
6.4.2 处理命令行参数 .....	278	7.2.9 设定MySQL服务器连接参数 .....	351
6.4.3 把选项处理机制融合到MySQL客户 程序里 .....	285	7.2.10 调试 .....	354
6.5 查询的处理 .....	289	7.2.11 结果集元数据的使用 .....	358
6.5.1 处理无结果集的查询 .....	291	7.2.12 用DBI脚本来实现事务处理机制 .....	362
6.5.2 处理有结果集的查询 .....	292	7.3 DBI脚本实战 .....	364
6.5.3 一个通用的查询处理程序 .....	295	7.3.1 美国历史研究会: 生成会员名录 .....	364
6.5.4 另一种查询处理方案 .....	297	7.3.2 美国历史研究会: 发出会费催交 通知 .....	370
6.5.5 mysql_store_result()与mysql_use_result() 函数的对比 .....	298	7.3.3 美国历史研究会: 编辑会员记录项 .....	376
6.5.6 结果集元数据的使用 .....	300	7.3.4 美国历史研究会: 查找兴趣相同 的会员 .....	381
6.6 客户程序4——交互式查询程序 .....	305	7.3.5 美国历史研究会: 把会员名录 放到网上 .....	382
6.7 编写具备SSL支持的客户程序 .....	306	7.4 用DBI模块来开发Web应用 .....	385
6.8 嵌入式MySQL服务器程序开发库的使用 .....	311	7.4.1 配置Apache服务器来使用CGI脚本 .....	386
6.8.1 编写一个内建有嵌入式MySQL服务器 的应用程序 .....	311	7.4.2 CGI.pm模块简介 .....	388
6.8.2 生成一个内建有嵌入式MySQL服务器 的应用程序可执行二进制文件 .....	314	7.4.3 从Web脚本连接MySQL服务器 .....	394
6.9 其他论题 .....	314	7.4.4 基于Web的数据库浏览器 .....	397
6.9.1 在结果集上进行计算 .....	315	7.4.5 考试记分项目: 考试分数浏览器 .....	402
6.9.2 对查询命令中的特殊字符进行编码 .....	316	7.4.6 美国历史研究会: 查找兴趣相同 的会员 .....	405
6.9.3 对图像数据进行处理 .....	318	第8章 MySQL应用程序设计接口:	
6.9.4 获取关于数据表结构的信息 .....	320	PHP语言 .....	410
6.9.5 MySQL程序设计工作中的常见错误及 预防办法 .....	320	8.1 PHP语言概述 .....	411
第7章 MySQL应用程序设计接口:		8.1.1 函数与include文件的使用 .....	418
Perl DBI .....	324	8.1.2 一个简单的数据检索页面 .....	422
		8.1.3 对查询结果进行处理 .....	425



8.1.4 返回结果里NULL值的检测 .....	431	10.3.4 重新安置一个数据库 .....	485
8.1.5 出错处理 .....	432	10.3.5 重新安置一个数据表 .....	487
8.1.6 引号问题 .....	434	10.3.6 重新安置InnoDB表空间 .....	487
8.2 PHP脚本实战 .....	435	10.3.7 重新安置状态文件和日志文件 .....	488
8.2.1 考试记分项目：考试分数的录入 .....	436	第11章 MySQL数据库系统的日常管理 .....	489
8.2.2 美国历史研究会：总统生平小测验 .....	449	11.1 新MySQL软件的安全措施 .....	489
8.2.3 美国历史研究会：会员个人资料的在 线修改 .....	454	11.1.1 权限表的初始设置情况是怎样的 .....	490
 <b>第三部分 MySQL系统管理</b>		11.1.2 为MySQL初始账户设置口令 .....	491
第9章 MySQL系统管理简介 .....	464	11.1.3 为第二个MySQL服务器设置口令 .....	493
9.1 管理职责概述 .....	464	11.2 安排MySQL服务器的启动和关闭 .....	494
9.2 日常管理 .....	465	11.2.1 在UNIX系统上运行MySQL服务器 .....	494
9.3 安全问题 .....	466	11.2.2 在Windows系统上运行MySQL 服务器 .....	498
9.4 数据库修复和维护 .....	467	11.2.3 设定MySQL服务器的启动选项 .....	500
第10章 MySQL的数据目录 .....	468	11.2.4 关闭服务器 .....	501
10.1 数据目录的位置 .....	468	11.2.5 在连接不上MySQL服务器时重新获得 对服务器的控制 .....	502
10.2 数据目录的结构 .....	471	11.3 管理MySQL用户账户 .....	504
10.2.1 MySQL服务器如何提供对数据 的访问 .....	472	11.3.1 创建MySQL用户账户并进行授权 .....	505
10.2.2 MySQL数据库在文件系统里 如何表示 .....	473	11.3.2 收回权限和删除用户 .....	514
10.2.3 MySQL数据表在文件系统里 如何表示 .....	474	11.3.3 修改口令或重新设置丢失的口令 .....	515
10.2.4 SQL语句如何映射为数据表 文件操作 .....	475	11.4 维护日志文件 .....	515
10.2.5 操作系统对数据库和数据表命名 的限制 .....	476	11.4.1 常规查询日志 .....	518
10.2.6 影响数据表最大尺寸的因素 .....	478	11.4.2 慢查询日志 .....	518
10.2.7 数据目录的结构对系统性能 的影响 .....	479	11.4.3 变更日志 .....	518
10.2.8 MySQL状态文件和日志文件 .....	480	11.4.4 二进制变更日志和二进制日志 索引文件 .....	519
10.3 重新安置数据目录的内容 .....	483	11.4.5 错误日志 .....	520
10.3.1 重新安置方法 .....	483	11.4.6 日志文件的失效处理 .....	521
10.3.2 评估重新安置的效果 .....	484	11.5 其他MySQL服务器配置问题 .....	527
10.3.3 重新安置整个数据目录 .....	485	11.5.1 对MySQL服务器的连接监听情况 进行控制 .....	527
		11.5.2 激活或者禁用LOAD DATA语句的 LOCAL能力 .....	528
		11.5.3 国际化和本地化问题 .....	528
		11.5.4 选择数据表处理程序 .....	531
		11.5.5 配置InnoDB表空间 .....	532

11.5.6 优化MySQL服务器 .....	536
11.6 运行多个MySQL服务器 .....	540
11.6.1 运行多个MySQL服务器需要注意 的问题 .....	541
11.6.2 配置和编译不同的MySQL服务器 ...	543
11.6.3 设定MySQL服务器启动选项 的策略 .....	544
11.6.4 用mysqld_multi脚本来启动多个 MySQL服务器 .....	545
11.6.5 在Windows系统上运行多个 MySQL服务器 .....	547
11.7 设置镜像服务器 .....	549
11.7.1 镜像机制概念 .....	549
11.7.2 建立主-从镜像关系 .....	550
11.8 升级MySQL软件 .....	553
第12章 MySQL安全技术 .....	556
12.1 内部安全性:防止未经授权的文件 系统访问 .....	556
12.1.1 如何偷取数据 .....	557
12.1.2 保护你的MySQL安装程序 .....	558
12.2 外部安全性:防止未经授权的 网络访问 .....	564
12.2.1 MySQL权限表的结构和内容 .....	564
12.2.2 MySQL服务器如何对客户进行 访问控制 .....	571
12.2.3 一个与权限有关的难题 .....	576
12.2.4 应该避免的权限表风险 .....	579

12.2.5	不用GRANT语句创建MySQL 账户.....	587
12.3	建立加密连接 .....	583
第13章	MySQL数据库的备份、维护 和修复 .....	587
13.1	与MySQL服务器进行协调 .....	588
13.1.1	使用内部锁定机制防止两个操作 相互干扰.....	589
13.1.2	使用外部锁定机制防止两个操作 相互干扰.....	592
13.2	在灾难发生前做好准备工作 .....	593
13.2.1	充分利用MySQL服务器的自动 恢复能力.....	593
13.2.2	备份和拷贝数据库.....	594
13.3	数据表修复和数据恢复 .....	602
13.3.1	检查和修复数据表.....	602
13.3.2	使用备份恢复数据.....	609

#### 第四部分 附 录

附录A	获得并安装有关软件 .....	616
附录B	数据列类型指南 .....	633
附录C	操作符与函数用法指南 .....	643
附录D	SQL语法指南 .....	694
附录E	MySQL程序使用指南 .....	769
附录F	C API指南 .....	829
附录G	Perl DBI API指南 .....	864
附录H	PHP API指南 .....	891
附录I	挑选ISP .....	922



# 第一部分 MySQL 基础知识

## 第1章 MySQL 和 SQL 入门

## 第2章 MySQL 数据库里的数据

## 第3章 MySQL SQL 语法及其使用

## 第4章 查询优化

# 第1章 MySQL和SQL入门

这一章主要介绍MySQL关系数据库管理系统（relational database management system, RDBMS）及MySQL所使用的结构化查询语言（Structured Query Language, SQL）的基础知识。在这一章里，我们对大家应该掌握的术语和概念进行了解释，对书中示例所用到的样板数据库进行了描述，还对MySQL数据库的创建和交互进行了一些介绍。

如果你在数据库方面是一个新手，不能肯定自己是否需要或者能否用得上数据库，就应该从这一章开始学习。如果你对MySQL或SQL只是早有耳闻却知之不详，也应该从作为入门指南的这一章入手。已经具备一定的MySQL或其他数据库系统使用经验的读者可以跳过这一章内容。但希望大家至少要看看1.2节，熟悉一下sampdb数据库的结构与内容，因为该示例将贯穿全书。

## 1.1 MySQL概述

本节将描述一些能够证明MySQL数据库系统很有用处的事例，让大家了解MySQL都能用来做些什么事情以及它们会对你的工作有什么样的促进和帮助作用。如果你用不着这些事例就已经信服了数据库的作用——也许你心里正有一个难题，因而急于让MySQL运转起来以解决之——不妨立刻前进到1.2节。

从本质上讲，数据库系统只不过是一套对大量信息进行管理的办法而已。信息可以来自各种来源。比如说，我们可以用它来代表科研数据、商业账目记录、客户订单、体育比赛成绩、销售业绩报告、个人爱好资料、人事档案记录、bug（程序漏洞）报告、学生考试成绩，等等。总之，数据库系统有着处理大范围信息的能力，但单纯因为其本身的缘故而安装并使用一个这样的系统却不见得有必要。假如某项工作已经有了很好的解决方案，而你却在“为使用而使用”的心理驱使下引入了一个数据库系统，那就显得太不明智了。超市采购清单就是一个很好的例子：在出发前，先把自己想买的东西列在一张纸上；到超市后，每买到一样东西，就把它从清单上划掉；而等采购完毕时，这张纸就大可以丢掉不要了。几乎没有人会因为这种事情去动用数据库。如果你拥有掌上电脑，那我就敢这样说：你可能会用掌上电脑的记事本功能来处理超市采购清单，而不会使用其数据库功能来做这件事。

当需要组织和管理的信息量很大或者信息本身很复杂的时候，仍依靠手工去处理有关的数据记录就会变得越来越力不从心。对那些每天要达成并处理上百万项交易的大公司来说，数据库是一件不可或缺的东西。而有些小规模的项目也很值得用一个数据库，校友录就是这方面的一个很好的例子。虽然校友录的信息量并不算大，涉及到的操作种类也不算多，但如果不使用数据库的话，它很快就会变得难以管理。再来看看以下几种场景：

- 你开了一家木工厂并雇用了几名员工。需要有一份员工名单和一份工资表，把何时何地给哪些员工发过工资的事记下来，还得把工资总数加起来以便向政府有关部门报税。还需要

把工厂所接的每一单木工活、每单木工活都是由哪几位员工负责完成的等事情记录下来。

- 你开了一家汽车配件连锁店。为了满足顾客的订单，需要随时了解某个零件在哪一家分店还有库存。
- 作为一名玩具销售商，必须格外关心流行趋势。你需要通过每天的销售和库存情况来决定是增加库存（某种玩具正变得越来越流行）还是减少库存（这样就不会有卖得不好的玩具压手里）。
- 你在长年的科研工作中积累了大量的数据。为了发表研究成果，必须对这些数据进行筛选和分析。这是一个沙里淘金般的工作，需要从数量巨大的原始数据里筛选出一些典型的数据来进行统计学分析，再根据分析结果推导出结论来。
- 你是一个广受欢迎的演讲人，旅行各地并参加各种集会，比如毕业典礼、商业会议、社团集会、政治演讲等等。你做了那么多次的演讲，结果自己都记不清各地的演讲主题了。为了安排好今后的演讲计划，你想把自己以前的演讲记录都整理出来。当再次回到自己曾经做过演讲的某个地方时，你可不想当众重复一遍上次的演讲内容，一份准确的演讲记录将帮助你避免这种尴尬。你还想把自己演讲的受欢迎程度记下来（你在“大都会爱犬俱乐部”做的题为“我爱猫咪的理由”的演讲基本上是一场彻底的失败，而你并不想在下次到那个地方时再重蹈覆辙）。
- 你是一位老师，需要记录学生的考试成绩和出勤情况。每进行一次考试或测验，你就会把学生们的成绩记录下来。把成绩记到成绩本上并不复杂，但今后想对这些成绩进行分析可就麻烦了。对各次考试的成绩进行排序以确定分数线、学期结束时为每名学生计算总评成绩、统计学生们的出勤情况等等，如果以人工方式来完成这些工作的话，那未免也太落后于这个时代了。
- 你在某机构里担任秘书一职，负责维护和管理该机构的会员名录。（这个机构可以是专业团体、俱乐部、收藏公司、交响乐团、健身俱乐部等等。）你每年都要打印一份会员名录，这份名录的依据是一份字处理文档；每当会员资料发生变化时，你就得用字处理软件来修改这份文档。因为字处理文档的缘故，你的很多好想法都无法实现，所以你对目前的状况感到很厌倦。很难对会员名录进行多种排序；很难从中选出指定的部分（比如只列出人名和电话号码）；很难把符合某种条件的会员（比如需要续交会员费的那些会员）都找出来，所以你每个月都得依靠自己去把这些会员找出来并给他们寄去续会通知。此外，你还希望增加一个人来分担由你负责的会员名录修改工作，但该机构没有足够的预算，根本不可能再雇一个人。你听说过“无纸办公”，知道它指的是电子化的办公手段，但你并不了解它对你有什么好处：会员名录已经是电子化的了，可具有讽刺意味的是，除了把名录打印成册以外，任何其他类型的工作都不容易完成。

在上面列举的这些场景里，有的信息量很大，有的信息量则相对很少。但它们有一个共同的特点，即这些事务原先都是以手工方式完成的，但引入一个数据库系统将大幅度提高工作效率。

那么，诸如MySQL之类的数据库系统会给你带来哪些特别的好处呢？这要看具体需要和想达到的目标到底是什么——在上面这些例子里，不同的场景就有着不同的目标。下面，我们将



以一个大家都很熟悉的典型事物为例来说明数据库的作用：数据库管理系统通常被人们用来取代文件柜，而事实上，数据库系统的作用也正像一个巨大的文件柜那样，只是它里面已经有很多预先建立好的信息记录管理功能罢了。与手工方式相比，以电子化手段来管理信息的优越性是非常明显的，同时也是非常重要的。我们来看一个例子，假设你所在的办公室需要负责管理公司的顾客资料，以下就是一些MySQL的信息记录管理功能给你带来的巨大帮助：

- **缩短信息记录的录入时间。**当需要添加一项新的信息记录时，用不着拉开文件柜的各个抽屉以确定需要把这条记录添加到什么地方。只需把这条记录提交给资料管理系统就行了，把该记录存放到适当位置的工作将由资料管理系统去完成。
- **缩短信息记录的检索时间。**当需要查找某条信息记录时，用不着拉开文件柜的各个抽屉就能找到想要的资料。假设你工作在一个牙科诊所。如果你想给最近没来参加定期检查的人们发一封提醒信，就完全可以让资料管理系统去把这些人的记录资料查找出来。当然，这与你告诉另一位员工说“请把最近6个月没来参加定期检查的人查找出来”的情况是不同的。利用数据库，就可以直接用下面这条看起来很奇怪的语句完成这项工作：

```
SELECT last_name, first_name, last_visit FROM patient
WHERE last_visit < DATE_SUB(CURDATE(),INTERVAL 6 MONTH);
```

这条语句看起来挺唬人，尤其是在你从没见过类似东西的情况下。可与以前需要花费好几小时的时间才能把有关结果查找出来的情况相比，现在你只需花费一两秒钟的时间就能完成这项工作，这一点还是相当吸引人的。（现在，请不要被这条奇怪的语句吓倒。用不了多长时间，你就不再会对它感到陌生了。事实上，等你学习完这一章的内容，就会明白这条语句到底有什么含义了。）

- **灵活的信息检索顺序。**用不着按照当初存储有关信息记录的顺序（比如按患者的姓氏顺序）来检索它们。可以让资料管理系统按所希望的任意顺序来把有关的信息记录找出来：按姓名也行，按医疗保险公司名称的顺序也行，按最近一次的就诊时间顺序也行，等等。
- **灵活的输出格式。**把想要的资料查出来以后，不必再把它们手工抄写下来。可以让资料管理系统把它们生成为一份名单。有时候，可能需要把有关信息打印出来；有时候，可能想把它们用在另一个程序里。（比如说，在生成一份最近没来参加定期诊断的患者名单后，可以把这些资料送到文字处理软件里去打印催诊通知，然后再给那些患者寄去。）也许你只对汇总信息（比如总共有多少人没来参加定期诊断）感兴趣。有了数据库，就用不着再亲自去统计这些人数了，资料管理系统会轻而易举地生成这些汇总信息。
- **信息记录能同时被多名员工使用。**在“有纸办公”的年代，如果有两个人同时需要查看同一份资料，第二个人就必须等第一个人把资料放回原处之后才能拿到它。有了MySQL，就能让多位员工同时使用同一份资料了。
- **信息记录的远程访问和电子化传输。**“有纸办公”只允许在信息资料的存放地点使用它们或者是让别人给你复印一份送过来。电子化的信息记录将允许在远程对这些记录进行访问或者通过电子化手段进行传输——如果你的牙科诊所开设有分支机构，分支机构里的医护人员能从他们自己的办公地点来存取有关的资料，你不再需要通过信使来传送这些资料。如果别人想获得有关记录却没有与你同样的数据库软件，那么，只要他能使用电子邮件，

你就可以把他想要的资料查找出来并通过电子化手段传送给他。

如果你以前曾经使用过数据库管理系统，就会亲身体会到上面列举的种种好处，而你现在的想法可能已经超越“电子化文件柜”的范畴了。很多企事业单位现在都把数据库与Web站点结合起来使用，这类情况是一个很好的例子。现在，假设你所在的公司有一个库存商品数据库，每当有顾客打电话来询问你们公司的仓库里是否存有某种货物以及价格时，销售人员就会利用这个数据库来进行销售。但这只是数据库比较传统的用法。如果你们公司还有一个可供顾客访问的Web站点，就可以增加一项新的服务项目——为顾客提供一个查询页面，让他们自己去查询商品库存情况和价格信息。你的顾客迅速地获得了他们想要的资料，这些资料是从你的数据库里查出来的，而你为他们提供的库存商品查询功能又是自动完成的。顾客立刻获得了资料，不必再拿着电话筒听占线忙音，也不必再受你们公司上下班时间的限制。而每一位使用你们公司Web站点的顾客都替你省下了一小笔需要支付给电话接听人员的工资。（你可以把如此节省下来的钱支付给Web站点上的员工，不知这是不是个好主意？）

你还可以更进一步地发挥数据库的作用。基于Web的库存查询功能不仅能满足顾客的需求，而且对你们公司也有着莫大的好处。这些查询本身能让你了解顾客都想买哪些商品，而查询结果则能让你了解你是否能够满足顾客的要求。如果你的仓库里没有顾客想要的东西，你可能会失去这笔生意。所以把来自顾客的库存查询信息（他们想买什么、你是否有足够的库存等等）记录下来的做法不失为一个好主意。可以根据这些信息来调整库存，为顾客提供更好的服务。

基于Web的数据库应用还能提供另一种服务：在Web页面上提供广告。虽然我和你一样都不喜欢这类东西，但MySQL能为你提供这类时髦的服务项目却是一个不争的事实——你完全可以把广告保存在Web服务器里并在有顾客访问时把它们显示在对方的浏览器上。同时，MySQL还可以对这种广告宣传活动进行跟踪，把顾客都看过哪些广告、看到了多少次、顾客是通过哪些站点访问过来的等信息记录下来。

说了半天，MySQL到底是怎样工作的呢？寻找这一答案的最佳办法是亲身体会一下。为了达到这一目的，我们需要先建立一个样板数据库。

## 1.2 样板数据库

本节对将在本书后续各章中使用的两个样板数据库进行描述。在学习MySQL使用方法的过程中，它们将充当各有关示例的信息源。这两个样板数据库是根据此前介绍的以下两种场景总结出来的：

- **你担任某机构秘书一职的场景。**我们需要对“某机构”做出更明确的定义，假设它的特点如下：这个机构是由一些对美国历史感兴趣的人自发地组织起来的（我们不妨把这个机构称为“美国历史研究会”）。出于个人的爱好，那些会员将自愿定期交纳一定的会费以维持其会员身份。会员交纳上来的会费将用于支付研究会的开支，比如会员刊物*Chronicles of U.S. Past*（美国历史年表）的印刷费用。这个研究会建有一个小规模Web站点，但这个站点目前还没有得到充分的开发和利用。就眼下来说，这个站点还只能用来发布一些基本的信息，比如这个研究会是什么的、它有哪些官员、怎样加入这个研究会等等。

- **考试分数记录场景。**在学期中，你的日常工作是负责考试与测验、记录考试分数、给学生打学分。在学期末，你将对学生的成绩做出总评，并把学生们的总评成绩和出勤情况上报给校方。

下面，我们将对这两种场景做进一步的分析：

- 你需要决定自己想从数据库得到哪些东西——也就是想达到的目标。
- 你需要决定自己想把哪些东西放到数据库里——也就是想记录哪些信息。

我们把“想从数据库得到哪些东西”放在了“想把哪些东西放到数据库里”的前面，这看起来似乎有些本末倒置——无论如何，得先把数据放到数据库里去才能对它们进行检索。事情是这样的，如何使用数据库主要取决于想达到的目标，而目标又主要取决于将从数据库检索出来的东西而不是放到数据库里的东西。如果今后根本不打算使用放入数据库里的某些信息，那就用不着浪费自己的时间和精力而把它们放进去。

### 1.2.1 美国历史研究会

在这个场景里，你担任着这个研究会的秘书职务。眼下，你正使用一份字处理文档来管理着这个研究会的会员名录。客观地说，利用这份字处理文档来打印一份全体会员名录的工作还是很容易完成的，可如果还想利用它再做些别的事情就没那么容易了。主要有以下几个目标：

- 希望能够根据不同情况把会员名录输出为其他格式，只输出符合特定要求的资料。首先，每年生成一份全体会员的名录——这是研究会的传统工作之一，而你还想继续坚持下去。另外，还想利用会员名录做一些其他工作，比如向研究会提供一份最新的会员名单以便邀请他们出席研究会的周年宴会。完成这些工作所需要用到的信息是有所不同的：全体会员名录需要用到每位会员的完整资料，而周年宴会只需用到会员的姓名（这项工作可不容易用字处理文档来完成）。
- 希望能够根据各种限制条件有选择地找出部分会员来。比如说，想知道近期有哪些会员需要续交年费以保留其会员资格。另外，还希望能够根据一些关键词把会员们分别组织成一个一个的讨论组，这些关键词代表着每位会员对美国历史上的不同时期的兴趣，比如，Civil War（独立战争）、Depression（大萧条）、civil right（民权）、Thomas Jefferson（托马斯·杰弗逊）总统的生平事迹等等。很多会员希望你能为他们提供一份与自己志趣相同的其他会员的名单，而你也非常想满足这些会员的愿望。
- 希望把研究会的会员名录发布到研究会的Web站点上去，这对会员和你都有好处。如果能通过某种自动化的过程把会员名录转换为Web页面的话，就能以比纸张形式更有时效的方式来管理会员名录的在线版本了。如果这份在线名录还支持检索功能的话，会员们就能轻松地自行查找他们感兴趣的信息了。比如说，如果某位会员想知道还有哪些会员也对Civil War（独立战争）感兴趣，他完全可以自己去进行查询，用不着再麻烦你去帮他查找有关资料了，而你也能抽出时间去做些别的事情。

必须承认，数据库并不是世界上最令人激动的东西，所以我也不打算鼓吹说使用数据库能够激发人们的创造性思维。不过，如果你不是站在与信息进行角斗的立场上（你与原来那份字



处理文档之间可能是一种这样的关系),而是把它们当做一种能够以一些更简易有效的方式加以利用的东西(希望你在使用了MySQL以后能够这样想),就肯定会发现很多使用这些信息的新方法:

- 如果会员名录能够以在线名录的形式添加到Web站点上,这些信息流肯定会发挥出其他的作用。比如说,如果会员能够以在线方式自行修改本人的资料并刷新数据库,就用不着再由你来负责有关资料的录入工作了。同时,会员资料的时效性和准确性肯定会有一个很大的提高。
- 如果把电子邮件地址也添加到数据库里,就可以通过电子邮件来提醒会员更新自己的个人资料。在发给会员的邮件里,可以附上他们的现有资料,让他们自己去核查这些信息并通过研究会Web站点提供的有关功能进行必要的修改。
- 数据库将大大拓展研究会Web站点的用途,而这些拓展并不仅仅局限于会员名录。研究会自己的会刊Chronicles of U.S. Past(美国历史年表),每期会刊里都有一个专为儿童准备的历史知识测验栏目。最近几期会刊里的小测验题目都集中在美国总统的生平事迹方面。你在研究会的Web站点上也可以开设一个类似的儿童栏目并以在线方式给出历史知识小测验题目来。这个栏目甚至可以搞成互动形式——小测验的题目都是由Web服务器以实时方式直接从数据库里提取出来并展示给站点访客的。

太让人兴奋了!所想到的这些数据库应用让你觉得自己还有能力再往前迈出好几步。在重新回到现实里来之后,开始考虑以下一些实际的问题:

- **这是不是有点野心勃勃?**要想把这些事都建立起来,工作量是否太大?要知道,空想总是要比实干容易,我也不想虚伪地告诉大家说这些设想都很容易实现。不过,等你学习完这本书的时候,现在勾勒出来的这些设想将全部成为现实。请记住:不必把这些设想一次性地全都实现出来,把这些工作分成一个一个的步骤,然后再一个一个地让它们变为现实。
- **MySQL能够胜任所有这些工作吗?**不,它不能,至少单靠它自己不能。比如说,MySQL不具备直接开发Web程序的功能。仅仅依靠MySQL是不能完成这里讨论的所有工作的,必须把它与其他软件开发工具结合起来才能完善和扩展它的功能。

我们将利用Perl脚本程序设计语言和Perl语言中的DBI(database interface,数据库接口)模块来编写用来访问MySQL数据库的脚本程序。Perl有着强大的文本处理功能,能够对数据库的查询结果进行极其灵活的处理并生成各式各样的输出。比如说,可以用Perl生成一份RTF(Rich Text Format)格式的会员名录,而任何一种文字处理软件都能识别出这种格式。我们还将用到另一种脚本程序设计语言PHP。PHP特别适合用来编写Web应用,同时它也很容易与数据库进行合作。这将使你能够从Web页面来正确地运行MySQL查询,并把数据库的查询结果包括在新生成的页面里。PHP与Apache(世界上最流行的Web服务器软件)配合得非常好,用它来提交查询表单并显示查询结果是很容易的。

MySQL能够非常好地与这些工具集成在一起,可以灵活组合这些开发工具以达成目标。有些套装开发工具声称自己有着极高的“集成度”,被宣传得几乎是无所不能,可实际情况却是它们只能做到两两配合,多组件之间的配合并不理想;大家千万不要让自己陷在这样的泥坑里。

- 最后也是最重要的一个问题——这要花费多少钱？别忘了，你任职的美国历史研究会只有有限的预算。虽说有点让人难以置信，可采用上述组合的解决方案的确没有什么成本。如果你有一些关于数据库系统方面的常识的话，就该知道它们通常都要卖很高的价钱。与此形成鲜明对照的是，MySQL通常是免费的（详情请参阅MySQL Reference Manual）。我们将要使用的其他工具（Perl、DBI、PHP、Apache）也都是免费的。也就是说，即使把所有东西都算在一起，这个用途广泛的系统也不会让你花费很多的代价——除时间以外。

在哪种操作系统上来开发数据库的选择权由你来决定。这里介绍的软件开发工具全都能在UNIX（包括BSD UNIX、Linux、Mac OS X等）和Windows操作系统上使用，但某些专门针对UNIX或Windows的shell脚本或批处理脚本例外。

下面，再去看看另一场景中使用的样板数据库。

### 1.2.2 考试记分项目

在这个场景里，你是一位负责记录学生考试成绩的教师，想把手工记录的学生成绩簿转换到一个使用MySQL的电子化系统上去。在这个场景里，你想从数据库里得到的东西与目前使用学生成绩簿时的情况差不多：

- 每次考试或测验之后，你都要把学生的成绩记录下来。如果是考试，还需要对分数进行排序以评定相应的学分（A、B、C、D、F）。
- 在学期结束的时候，需要把学生这一学期的总分数计算出来，对这些总分数进行排序，再根据学生们的总分数评出他们的总学分。在计算总分数的時候，可能需要进行加权计算以区别考试成绩和测验成绩，因为考试的重要性通常要大于小测验的重要性。
- 在学期结束的时候，还要向校方提交学生的出勤情况。

目标是不再以人工方式对学生们的考试分数和出勤情况进行排序和汇总。换句话说，你希望每次考试后的分数排序工作以及各学期末的学生总评分和出勤情况的例行统计工作都能够交给MySQL去完成。为了实现这一目标，需要知道班级里的学生名单、他们每次考试或测验的分数以及考试当天缺勤的学生姓名。

### 1.2.3 关于样板数据库的说明

有些读者可能对“美国历史研究会”或者“考试记分项目”没有什么兴趣，希望这不会影响你继续研读本书。要知道，这些场景只是一些例子，它们本身并不是我们的学习目标。它们只是在学习使用MySQL及相关工具时需要用到的一种载体。只要稍微动点脑筋，就能看出这些样板数据库上的查询与你真正关心的具体问题之间的相似之处。假设你工作在前面提到的那个牙科诊所里。虽然在这本书里看不到多少与牙科医学有关的查询，但书中的很多查询都能运用到管理患者资料或办公室资料的工作中去。比如说，在“美国历史研究会”场景里，你需要把近期必须续费才能保持其会员资格的会员找出来；而这与你把近期需要来牙科诊所进行定期检查的患者找出来的情况是类似的——它们都是与日期有关的查询。因此，只要学会了如何写出一个用来找出哪些会员需要续费的查询，就可以把类似的原理运用到自己的具体工作中，并写

出一个用来找出哪些患者需要就诊的查询来。

## 1.3 数据库基本术语

读者可能已经注意到了，这虽然是一本介绍数据库的书，而且已经看了好多页了，但至今仍未遇到大量专业性的术语和技术词汇。事实上，到目前为止，虽然已经对样板数据库进行了粗略的描述，但我仍未描述过关于“数据库”应该是什么样子。可是，既然我们将要设计数据库并开始实现它，就无法继续回避有关的术语，而这正是本节将要介绍的内容。本节将对书中使用的一些术语进行解释，希望大家能够掌握它们的含义。值得庆幸的是，与关系数据库有关的概念大都比较简单。事实上，人们喜欢关系数据库的很大一部分原因就在于它们的基本概念都很简明易懂。

### 1.3.1 数据库的组织结构术语

在数据库的世界里，MySQL被划分到关系数据库管理系统（relational database management system, RDBMS）的范畴内。我们可以把这个短语划分为以下几个部分：

- “数据库”（database，即RDBMS里的“DB”）就是用来存放信息的资料库，它们的构造既简单又遵守一定的规律：
  - 数据库里的数据都存放在数据表（table）里。
  - 数据表是由数据行（row）和数据列（column）构成的。
  - 一个数据行就是数据表里的一条记录（record）。
  - 记录可以包含多个信息项，数据表里的每一个数据列都对应着一个这样的信息项。
- “管理系统”（management system，即RDBMS里的“MS”）指的是通过插入、检索、修改、删除记录等对数据进行操作的软件。
- “关系”（relational，即RDBMS里的“R”）表示它是DBMS中的一种，这种DBMS把存放在某个数据表里的信息与存放在另一个数据表里的信息“关联”（即相互匹配）起来，而这种关联是通过查找两个数据表有无共同的元素而实现的。关系数据库管理系统的威力在于它能方便地抽取出数据表里的数据并把它们与来自其他相关数据表的信息结合起来，为那些单独利用某个数据表无法找到答案的问题找到答案。

关系数据库是如何把数据组织到数据表里的？又是如何把来自不同数据表的信息关联在一起的呢？我们来看一个例子。假设你有一个Web站点，在站点上提供了广告服务，并与一些想让访问站点的网页浏览者看到其广告的公司签订了服务合同。每当有访客点击了某个页面时，你就会把一条广告嵌入到相应的页面里，然后再把嵌有广告的页面发送给访客的浏览器；同时，你还会向刊登这条广告的公司收取一笔小小的费用。为了记录这些信息，使用了三个数据表（如图1-1所示）。company（公司）数据表由以下几个数据列构成：公司名称（company\_name）、公司编号（company\_num）、地址（address）、电话号码（phone）。ad（广告）数据表由以下几个数据列构成：广告编号（ad\_num）、“拥有”该广告的公司的编号（company\_num）、每点击一次该广告的计费标准（hit\_fee）。hit（点击情况）数据表由以下几个数据列构成：广告编号



(ad\_num)、该广告被发送给浏览者的日期(date)。

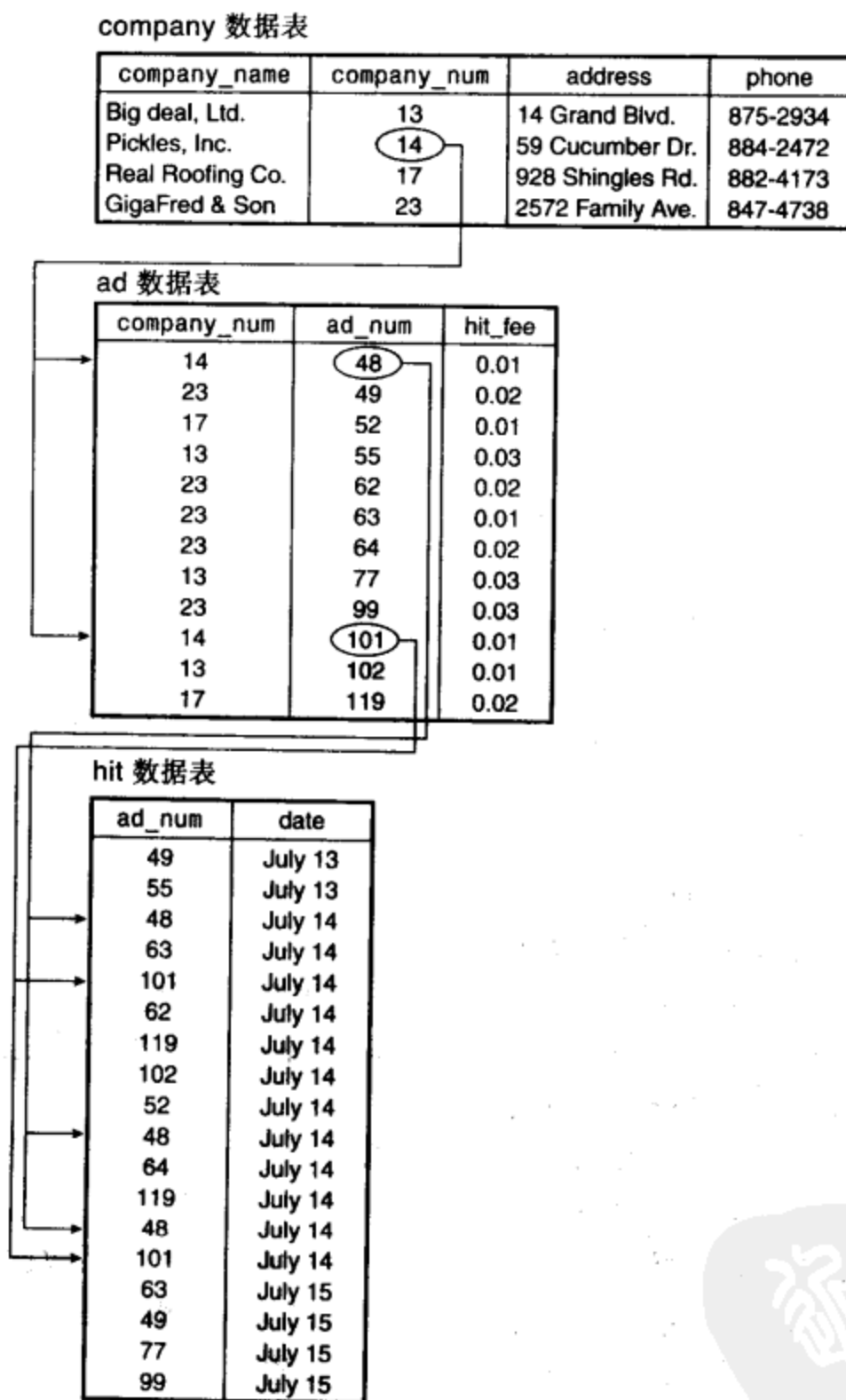


图1-1 网站广告服务数据表

有些问题利用单个数据表就能做出回答。比如说，如果想知道有多少家公司与你签订了广告服务合同，只要数一数company数据表总共有多少行就行了。如果想知道在某个给定的时间段内有多少次广告点击，也只需用到hit数据表。可有些问题比较复杂，需要查询多个数据表才能找出答案。例如，在7月14号这一天里，Pickles公司的各条广告分别被点击了多少次？要想回答出这个问题，就必须把这三个数据表都用上，如下所示：

- 1) 在company数据表里根据公司名称（Pickles公司）查出对应的公司编号（14）。
- 2) 利用这个公司编号在ad数据表里查找与之匹配的记录以确定有关的广告编号。这里找到

了两个符合条件的广告，它们的编号是“48”和“101”。

3) 利用这两个广告编号从hit数据表里把给定日期范围内的匹配记录找出来，再对匹配的记录个数进行统计。最后，查出编号为“48”的广告有3个匹配，编号为“101”的广告有2个匹配。

你可能会觉得这太复杂了，但这正是关系数据库系统所擅长的。而且，虽然看起来很复杂，但上面这几个步骤也只是几个简单的匹配操作而已——把一个数据表与另一个数据表关联起来，看前一个数据表的数据行取值是否与后一个数据表的数据行取值相匹配。把这几步简单的操作推广开来，就能找出各种问题的答案：各家公司分别有多少个不同的广告？哪家公司的广告最受欢迎？每个广告会带来多少利润？在本结算周期内，每家公司应该支付你多少广告费？

现在，你对关系数据库理论的了解已经足以让你读懂本书后续章节的内容了，这里不再介绍“第三范式”（Third Normal Form）、“实体关系图”（Entity Relationship Diagram）之类的枯燥概念。（如果读者想了解这些概念，建议阅读C. J. Date或E. F. Codd的著作。）

### 1.3.2 数据库查询语言术语

为了与MySQL进行“交谈”，需要使用一种名为SQL（Structured Query Language，结构化查询语言）的计算机语言。SQL是现时期的标准数据库语言，在各种主流的数据库系统上都能使用。SQL中的各种语句使它能够高效率地与数据库进行互动。

与其他计算机语言一样，初次接触SQL的人往往会觉得它很奇怪。比如说，在创建数据表的时候，必须告诉MySQL这个数据表的结构是什么样的。很多人会把数据表想像成一个表格或者一幅图画，但MySQL却不这么想，所以在创建数据表的时候必须写出下面这样的代码：

```
CREATE TABLE company
(
    company_name CHAR(30),
    company_num INT,
    address CHAR(30),
    phone CHAR(12)
);
```

这类语句往往会让SQL新手产生畏惧感，但熟练运用SQL却并非一件只有程序员才能掌握的难事。随着对这种语言的熟悉程度的加深，你对CREATE TABLE这类语句的看法也会悄悄地发生转变——它们将不再是一团难以理清的乱麻，而是能帮助你信息做出描述的工具。

### 1.3.3 MySQL的体系结构术语

MySQL采用的是客户/服务器体系结构。因此，当使用MySQL的时候，实际是在使用两个程序：

- 所谓MySQL服务器指的是mysqld程序，它运行在存放着数据库的机器上，负责在网络上监听并处理来自客户的服务请求，根据这些请求去存取数据库的内容，再把有关信息回传给客户。

- 所谓MySQL客户是这样一些程序：它们负责连接到数据库服务器，并通过向服务器发出查询命令告诉服务器它们需要哪些信息。

MySQL的发行版本包括数据库服务器和几个客户程序。可以根据自己的具体情况来选用一种客户程序。mysql是人们最常用的客户程序，它是一个交互式的客户程序，通过它发出查询命令并查看结果。mysqldump和mysqladmin是两种主要用于数据库管理工作的客户程序，前者用来把数据表的内容导出到一个文件里，后者用来检查数据库服务器的工作状态和进行一些数据库管理方面的操作，比如通知数据库服务器停止运行等。MySQL发行版本里往往还会有其他的客户程序，如果MySQL自带的客户程序不能满足应用要求，MySQL还提供有一个客户程序开发库，利用该开发库可以自行编写一些程序来解决问题。这个开发库可以从C语言程序里直接使用。如果你偏爱其他编程语言，还有Perl、PHP、Python、Java、C++、Ruby等程序设计接口可供选用。

MySQL的客户/服务器体系结构有以下一些好处：

- 并发控制（concurrency control）由服务器负责提供，因而不会出现两个用户同时修改同一条记录的现象。来自客户的请求全都要经过服务器，由服务器来安排它们得到处理的先后顺序。即使出现多个客户同时请求访问同一个数据表的情况，也用不着由这些客户去彼此找到对方并进行协商。它们只负责把请求发往服务器，而谁先谁后的事则完全由服务器去决定。
- 不必非得在存放着数据库的那台机器上进行登录。MySQL知道如何在因特网上运行，所以可以在任意地点运行一个MySQL客户程序，而这个客户程序能够通过网络连接到服务器。地理距离根本不是问题，可以从世界上任何一个角落来访问服务器。即使服务器位于澳大利亚而你带着一台笔记本电脑旅行到了冰岛，仍能访问自己的数据库。可这是否意味着别人也能通过因特网看到你的数据呢？答案是“不”。MySQL包括有一个灵活而且有效的安全系统，只有得到授权的人才能访问你的数据。而且，还可以进一步限制这些人只能做你允许他们做的事。比如说，财务部的Sally应该有查看和修改数据记录的权限，可售后服务部的Phil却只应该有查看它们的权限。总之，可以把这种访问权限控制细化到每一个人。从另一方面讲，如果只想拥有一个完全属于自己的系统，那么完全可以把访问权限设置成只允许客户程序从运行着服务器的那台主机上进行连接。

从MySQL 4开始，还可以以另一种方式来运行服务器程序。除原来以客户/服务器方式运行的mysqld服务器程序外，MySQL又新增了一个库函数形式的服务器libmysqld，可以把它链接到程序里以编写出独立的基于MySQL的应用程序。因为它被嵌入在一个独立的应用程序里，所以人们把libmysqld称为嵌入式服务器库（embedded server library）。嵌入式服务器与客户/服务器方案的主要区别是前者不要求主机里必须安装有MySQL软件。这使我们能够方便地制作出这样一种“自给自足”的应用软件包来：它们对外部操作环境的要求不高，与数据库有关的操作完全由它自己来负责完成（即使机器里没有安装MySQL软件也行）。这既是它的优点，也是它的局限性——如果机器里没有安装MySQL软件，其他软件包就无法访问该主机上的数据库了。



### MySQL与mysql之间的区别

请注意，MySQL指的是完整的MySQL RDBMS系统，而mysql则仅代表一个特定的客户程序（即该客户程序的名字）。它们的发音相同，但代表的却是不同的事物，所以本书要用大写和小写字母来区分它们。

说到发音，*MySQL Reference Manual*（MySQL参考手册）上给出的是“my-ess-queue-ell”。但SQL却有“sequel”和“ess-queue-ell”两种读法。怎么发音都无所谓，但迟早会有好心人来纠正你的发音。（我发音为“sequel”，这就是为什么我使用“a SQL query”而不是“an SQL query”。）

## 1.4 MySQL教程

好了，大家应该了解的预备知识也就是这么多。下面该让本书的主角MySQL上场了！

安排这样一个教程的目的是为了帮助大家熟悉MySQL的基本用法。这个教程是这样安排的：先把样板数据库和几个必要的数据库表创建出来，再通过样板数据库去学习如何对数据库表里的信息进行添加、检索、删除、修改等操作。通过这个教程，将学到以下技能：

- **MySQL数据库查询语言SQL的基本知识。**MySQL所使用的SQL语言与其他RDBMS使用的版本有着细微的差异。因此，即便你以前曾经接触过其他的RDBMS并有一定的SQL使用经验，也应该快速浏览一下本节的内容。
- **使用MySQL自带的标准客户程序与MySQL服务器进行通信。**在前面的内容里讲过，MySQL采用的是客户/服务器体系结构，服务器运行在存放着数据库的主机上，而客户端需要通过网络连接到服务器上。本教程的重点内容是mysql客户程序的使用方法，它负责读取你发出的SQL查询命令，把它们发送到服务器，再把经服务器处理后得到的查询结果显示给你。之所以把mysql作为介绍重点，是因为它能在MySQL支持的任何一种硬件平台上运行，是把你与数据库服务器直接联系在一起的纽带。教程中的某些示例还用到了另外两个客户程序：mysqlimport和mysqlshow。

书中所使用的样板数据库取名为sampdb，但某些读者可能需要另外给它起个名字。比如说，也许系统上已经有其他人把自己的数据库命名为sampdb了，或者MySQL管理员给这个样板数据库另外指定了一个名字。如果遇到这类情况，请把本书语句示例中的sampdb替换为具体使用的样板数据库名称。

即使系统中有多个用户都各自安装了样板数据库，出现在本书语句示例中的数据表名称也用不着做任何改动。在MySQL里，只要数据库的名字不重复，数据库里的数据表是允许同名的——MySQL自己会区分它们，不会因为分不清哪个对哪个而引起混乱。

### 1.4.1 获得样板数据库

在本教程的某些地方，用“样板数据库发行版本”（或“sampdb发行版本”，因为样板数据库的名字是sampdb）来指示有关文件。这些文件里存放着用来安装样板数据库的查询命令和数据，

它们的获取办法和安装步骤可以在附录A里查到。安装过程将自动创建一个名为sampdb的目录并把有关的文件存放到里面。建议在使用sampdb数据库练手之前，最好先切换到这个目录里。

### 1.4.2 最低配置要求

要想试用本教程中的示例，必须满足以下几项要求：

- 系统中已经安装了MySQL软件。
- 有一个用来连接数据库服务器的MySQL账户。
- 有一个专供练习使用的样板数据库。

MySQL客户程序和MySQL服务器是必不可少的。MySQL客户程序必须安装在自己能够动手操作的机器里；服务器可以安装在自己的机器里，也可以安装在别的主机里——只要有对它的连接权限，MySQL服务器可以放在任何地方。获得和安装MySQL的步骤请参考附录A。如果网络连接需要经过ISP（Internet Service Provider，因特网服务提供商），请提前查明该ISP提供的服务项目里有没有MySQL。如果没有这个服务项目，并且该ISP也不准备安装它，请更换ISP——附录I提供了一些关于如何挑选ISP的建议。

除MySQL软件外，还必须有一个MySQL账户，否则将无法连接MySQL服务器，也就无法创建样板数据库和有关的数据表。（如果读者已经有了一个MySQL账户，可以直接拿过来用，但建议另外申请一个专供学习本书时使用的账户。）

现在，我们遇到了一个“先有鸡还是先有蛋”的问题：要想申请一个用来连接MySQL服务器的账户，必须先连接到这个服务器上才行。一般来说，这需要在运行着MySQL服务器的主机上以MySQL的root用户身份登录，再用一条GRANT语句创建一个新MySQL账户。如果MySQL服务器就安装在自己的机器上并且正在运行，那么自己就能连接上服务器并创建一个新账户。在下面的示例里，给样板数据库sampdb建立了一个新的管理员账户，新账户的用户名是sampadm、口令是secret。（可以把它们改成别的，但要改就得把此处和本书其他有关内容里的用户名和口令都改过来。）

```
% mysql -p -u root
Enter password: *****
mysql> GRANT ALL ON sampdb.* TO 'sampadm'@'localhost' IDENTIFIED BY 'secret';
```

mysql命令的-p选项用于让mysql提示MySQL的root用户输入口令。所输入的口令将显示为一串星号字符，即示例中的“\*\*\*\*\*”部分。（这里假设已经为MySQL的root用户设置了口令。如果还没有给它设置口令，请在Enter Password:提示符后直接按下回车键。不过，root用户没有口令是很大的安全漏洞，应该尽快给它设置一个。）

如果打算今后就在运行着MySQL服务器的机器上连接MySQL，可以直接套用示例中的GRANT语句。示例中创建的新账户是一个数据库管理员账户，它不仅能让你以用户名sampadm加口令secret连接到服务器，而且还让你拥有sampdb数据库的全部访问权限。注意：GRANT语句并不能创建出数据库来，我们稍后再来讨论数据库的创建问题。

如果打算今后使用另一台计算机通过网络来连接MySQL服务器，就需要把示例中的localhost改为那台计算机的名字。举个例子，如果将从主机asp.snake.net来连接MySQL服务器，

就要把GRANT语句改写为下面这样：

```
mysql> GRANT ALL ON sampdb.* TO 'sampadm'@'asp.snake.net' IDENTIFIED BY 'secret';
```

如果你无法控制服务器，那就得求助于MySQL管理员来为你建立一个新账户了。如果是这种情况的话，需要把在本书各示例中出现的sampadm、secret、sampdb分别替换为MySQL管理员分配给你的用户名、口令和样板数据库名。

将在第11章对GRANT语句、建立MySQL用户账户、修改口令等问题做进一步讨论。

### 1.4.3 建立和断开与服务器的连接

通过UNIX系统的shell提示符或者通过Windows下的DOS控制台调用mysql程序，就能连接到MySQL服务器。这个命令如下所示：

```
% mysql options
```

在这本书里，提示符“%”表示有关命令可以在UNIX系统的shell提示符以及Windows系统的DOS提示符下使用。“%”是UNIX系统的一个标准提示符，另一个是“\$”。在Windows下，将看到像“C:\>”这样的提示符。

这个mysql命令行里的options部分表示允许是空白，但下面这种形式的mysql命令可能更常见一些：

```
% mysql -h host_name -p -u user_name
```

在调用mysql程序的时候，用不着把全部选项都写出来，但通常至少需要用户给出自己的用户名和口令。下面是这几个选项的含义和用法：

- -h *host\_name* (替换形式：--host = *host\_name*)

打算连接的服务器主机名。如果MySQL服务器就运行在正运行着mysql客户程序的同一台机器上，此选项就可以省略。

- -u *user\_name* (替换形式：--user = *user\_name*)

MySQL用户名。在UNIX系统上，如果MySQL用户名与登录名完全一样，就可以省略这个选项，mysql将自动把登录名用做MySQL用户名。

在Windows系统上，此选项的默认用户名是ODBC，但你可能不喜欢这个默认名。可以通过命令行上的-u选项明确地给出自己的MySQL用户名，也可以通过设置变量USER来给出用户名。比如说，可以用下面这条set命令设定一个名为sampadm的用户：

```
C:\> set USER=sampadm
```

如果把这条命令放到AUTOEXEC.BAT文件里，那么，每当启动Windows的时候，这条语句就将自动生效，并且不必在提示符下发出这条命令了。

- -p (替换形式：--password)

这个选项的作用是让mysql提示你输入MySQL口令。比如：

```
% mysql -h host_name -p -u user_name
```

```
Enter password:
```



当看到Enter password:提示符时,请输入口令。(输入的口令将不会显示在屏幕上,以免被站在身后的人偷看到。)注意:MySQL口令并不一定与UNIX或Windows口令相同。如果省略-p选项,mysql就将认为你不需要口令,也就不会提示你输入它了。

这个选项的另一种形式是在命令行上直接给出口令——以-p *your\_pass* (替换形式: --password=*your\_pass*, 其中*your\_pass*就是口令)的形式直接键入口令。但出于安全方面的考虑,建议大家最好别这样做——其他人可能会看到它。

如果确实想在命令行上直接敲入口令,有一点大家要特别注意:在-p选项和口令之间不允许有空格。-p选项的这一特点(不需要输入空格)很容易与-h和-u选项(需要空格)的情况弄混。

假设MySQL用户名和口令分别是sampadm和secret,那么,如果MySQL服务器运行在同一台主机上,就可以省略-h选项而像下面这样去连接服务器:

```
% mysql -p -u sampadm
Enter password: *****
```

在输入完这条命令后,mysql将显示Enter password:以提示输入口令。敲入口令(所输入的secret将在屏幕上显示为6个星号字符“\*\*\*\*\*”)。

如果一切正常,mysql将显示欢迎信息和一个mysql>提示符以表明它在等待发出SQL查询命令。下面是有关操作的完整过程:

```
% mysql -p -u sampadm
Enter password: *****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 7575 to server version: 4.0.4-log

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
mysql>
```

如果MySQL服务器运行在另一台机器上,就必须通过-h选项来指定那台机器的主机名。假设那台主机的名字是cobra.snake.net,则使用下面这样的命令:

```
% mysql -h cobra.snake.net -p -u sampadm
```

为简洁起见,如果没有特殊的需要,将在本书后面出现的mysql命令行上省略-h、-p和-u选项,但大家在做练习或实际工作中不要忘了输入它们。

在连接到MySQL服务器之后,随时都可以通过QUIT命令来结束这次会话。如下所示:

```
mysql> QUIT
Bye
```

也可以通过敲入\q或组合键Ctrl-D(适用于UNIX系统)来退出。

在刚开始学习MySQL的时候,很多人都觉得它的安全系统给自己添了不少麻烦——必须有足够的权限才能创建和访问数据库,必须正确地给出用户名和口令才能连接上服务器。但在抛开本书里的样板数据库而开始使用他们自己的数据记录之后,这些人的看法就会迅速转变为感激MySQL有这样一个能防止别人偷看或者(更糟糕)破坏自己信息的安防系统。

有些做法能让你把自己的账户资料提前设置好,从而不必在每次运行mysql的时候都去输入

一大堆的连接参数，将在第1.5节里对它们做详细介绍。简化服务器连接过程最常见的办法是把连接参数存放到一个选项文件里。如果想现在就建立一个这样的文件，不妨直接跳到第1.5节去看看。

#### 1.4.4 发出查询命令

连接上服务器以后，就可以发出查询命令了。我们将从这一小节开始向大家介绍一些mysql的基本使用情况。

利用mysql来进行数据库查询其实是一件非常简单的事情：先敲入有关命令，再在命令的末尾敲入一个分号(;)并按下回车键就行了。在输入了查询命令之后，查询命令将由mysql送往服务器去执行，服务器对查询进行处理并把其结果回送给mysql，最后由mysql把查询结果显示在屏幕上。

下面这个简单的查询命令将显示系统的当前日期和时间：

```
mysql> SELECT NOW();
+-----+
| NOW() |
+-----+
| 2002-09-01 13:54:24 |
+-----+
1 row in set (0.00 sec)
```

mysql将把本次查询的结果、构成本次查询结果的数据行的个数以及用来处理本次查询所花费的时间依次显示出来。为简洁起见，如无特殊需要，将在本书后面的示例里省略给出查询结果数据行总数与查询处理时间的行。

因为mysql必须以分号作为语句结束符，所以用不着把查询命令完整地写在同一个命令行上——可以用多个命令行来输入一条查询命令，如下所示：

```
mysql> SELECT NOW(),
-> USER(),
-> VERSION(),
-> ;
+-----+-----+-----+
| NOW() | USER() | VERSION() |
+-----+-----+-----+
| 2002-09-01 13:54:37 | sampadm@localhost | 4.0.4-beta-log |
+-----+-----+-----+
```

请注意，当输入了查询命令的第一行时，提示符将从mysql>改变为->。这是在提示你mysql认为你仍要继续输入查询命令。这种反馈非常重要——如果你遗漏了应该添加在查询命令末尾的分号，发生了变化的提示符将提醒你注意mysql仍在等待你继续进行输入。如若不然，就可能出现这样的情况：你在为MySQL经过了这么长的时间还没有完成查询而疑惑，而mysql却在耐心地等待你把这条查询命令输入完！（mysql还有几个其他提示符，将在附录E里介绍它们。）

在大多数情况下，查询命令允许以大写字母、小写字母或者大小写字母混用的形式来输入。比如，下面这几条查询命令是等效的：

```
SELECT USER();
select user();
SeLeCt UsEr();
```

在后面的示例语句里，将用大写字母来写出SQL关键字和函数名，用小写字母来写出数据库、数据表和数据列的名字。

如果想在查询命令里使用函数，请千万记住：在函数名与它后面的括号之间不允许出现空格。如下所示：

```
mysql> SELECT NOW ();
ERROR 1064: You have an error in your SQL syntax near '()' at line 1
mysql> SELECT NOW();
+-----+
| NOW() |
+-----+
| 2002-09-01 13:56:36 |
+-----+
```

这两个查询看起来差不多，但前面那个却是错误的，因为该命令里的函数名与它后面的括号之间多了一个空格。

结束查询命令的另一种做法是用\g来代替分号，如下所示：

```
mysql> SELECT NOW()\g
+-----+
| NOW() |
+-----+
| 2002-09-01 13:56:47 |
+-----+
```

还可以用\G来结束一条查询命令，这将使查询结果显示为竖直形式，如下所示：

```
mysql> SELECT NOW(), USER(), VERSION()\G
***** 1. row *****
      NOW(): 2002-09-01 13:56:58
      USER(): sampadm@localhost
      VERSION(): 4.0.4-beta-log
```

如果查询命令的输出行比较短，以\G作为查询命令结束符的效果还不太明显；但如果输出行比较长，比如每行查询结果会在屏幕上显示为几行的时候，\G结束符可以使屏幕输出内容更容易阅读。

如果已经输入了几行查询命令却不想执行它，可以敲入\c来清除（即取消）它，如下所示：

```
mysql> SELECT NOW(),
-> VERSION(),
-> \c
mysql>
```



请注意，提示符将变回为mysql>以表明mysql准备接收下一条查询命令。

可以把查询命令保存在一个文件里，再让mysql从文件中而不是键盘上读取有关的查询命令。这要用到shell（操作系统的命令解释器）的输入重定向功能。比如说，如果把查询命令保存在名为myfile.sql的文件里，就可以像下面这样来执行它们：

```
% mysql < myfile.sql
```

这个文件的名字可以随便起。我喜欢给它们加上.sql后缀以表明这个文件里存放的是SQL语句。

这种利用shell的重定向功能来执行mysql的做法特别适用于需要录入大量数据的情况，在第1.4.7节里，将用这种办法把数据录入到sampdb数据库里。与一条一条地手工敲入一大堆INSERT语句相比，让mysql从某个文件里读取它们要方便和快捷得多。

本教程的后续内容里有很多为大家进行练习而特意准备的查询命令，它们以mysql>提示符为标志且基本上都包括有查询结果。如果你输入的查询命令与示例中的一模一样，那查询结果也应该完全相同。但我在某些查询命令的前面没有给出提示符，它们主要用来阐明某个概念，不要求读者真的去执行它们。（当然，如果愿意，那么试试它们也没什么坏处。但如果打算用mysql来这样做的话，请不要忘记在它们的末尾加上一个分号作为结束符。）

#### 何时需要使用分号

本书中的大多数查询都以一个分号作为其结束符，这使得很容易看出它们到底在什么地方结束（尤其是在有多条查询命令的场合）。这与通过mysql来输入查询命令时的做法也是一致的。但是，分号并不是SQL语句的一个语法成分。换句话说，当在另一种上下文环境（比如从一个Perl或者PHP脚本）发出一条查询命令时，应该省略掉这个分号；否则，将可能看到一条出错信息。

#### 1.4.5 创建数据库

我们的学习将从创建sampdb样板数据库与其中的数据表、把有关数据录入各数据表、利用这些数据表里的数据完成一些简单的查询任务开始逐步展开。要想使用数据库，就必须经过以下几个步骤：

- 1) 创建（初始化）数据库。
- 2) 在数据库里创建各种数据表。
- 3) 对数据表里的数据进行插入、检索、修改、删除等操作。

数据库上最常见的操作是对现有数据进行检索；比较常见的操作是插入新数据、修改或者删除现有数据；比较不常见的操作是创建数据表；最不常见的操作是创建数据库。但因为我们的学习要从头开始，所以这里从最不常见的数据库创建操作入手，再经过创建数据表和录入原始数据等步骤之后到达最常见的数据库操作——数据的检索。

创建新数据库的方法是：先用mysql连接上服务器，再用一条CREATE DATABASE语句给出新数据库的名字：

```
mysql> CREATE DATABASE sampdb;
```

只有在创建出sampdb数据库之后，才能去创建该数据库里的各个数据表并对这些数据表的内容进行各种操作。

那么，创建一个数据库是否就把它选定为当前的默认数据库了呢？答案是“否”，可以用下面这条查询命令去核实一下：

```
mysql> SELECT DATABASE();
+-----+
| DATABASE() |
+-----+
|             |
+-----+
```

如果想把sampdb设置为当前的默认数据库，就需要发出一条USE语句：

```
mysql> USE sampdb;
mysql> SELECT DATABASE();
+-----+
| DATABASE() |
+-----+
| sampdb      |
+-----+
```

选定（打开）某个数据库的另一种方法是在调用mysql时在命令行上给出该数据库的名字：

```
% mysql sampdb
```

事实上，今后大家在选定数据库时用得最多的可能还是后一种方法。如果需要指定服务器连接参数，请把它们添加在数据库名字的前面。比如说，下面两条命令将把sampadm用户分别连接到本地主机和cobra.snake.net主机上的sampdb数据库上：

```
% mysql -p -u sampadm sampdb
% mysql -h cobra.snake.net -p -u sampadm sampdb
```

如果没有特别说明，以后的示例都假定在调用mysql时在命令行上命名sampdb数据库并使其成为当前的默认数据库。如果在调用mysql时忘了在命令行上命名数据库，请在mysql>提示符下发出一条USE sampdb语句。

#### 1.4.6 创建数据表

在这一小节里，将把样板数据库sampdb里的各个数据表创建出来。先来创建“美国历史研究会”场景所需要的各种数据表，然后再创建“考试记分项目”所需要的各种数据表。这是一些数据库教科书开始讨论“数据库的分析与设计”、“实体关系图”、“规范化过程”等概念的地方。这些概念当然有它们的出处和去处，但我更喜欢通俗地把它说成“数据库应该是什么样子”——它应该包含哪些数据表、各数据表应该有什么内容以及数据应该如何来表示。

这里所选定的数据表示方式并不是绝对的，换在另一种场合，可能会选用另一种方式来表示类似的数据——这应该由应用项目的具体要求以及有关数据的具体用途来决定。

## 1. “美国历史研究会”的数据表

“美国历史研究会”场景需要的数据表相当简单，主要包括：

- **president (总统) 数据表**——用来保存美国历届总统的描述性记录。我们将利用它在研究会的Web站点上提供历史知识在线小测验（会刊儿童栏目中的历史知识小测验的交互式电子化模拟）。
- **member (会员) 数据表**——用来保存每一位会员的个人最新资料。研究会将利用这个数据表来完成制作会员名录（纸印刷品以及在线版本）、自动提醒会员续交会费等有关工作。

### (1) president数据表

president数据表比较简单，所以我们先来讨论它。这个表里包含着关于美国历届总统生平的基本信息：

- **姓名**——在数据表里，有好几种办法可以用来表示姓名。比如说，既可以把姓名保存在一个数据列里，也可以把姓氏和名字分别保存在不同的数据列里。用一个数据列来保存姓名当然要简单一些，但这种做法有一定的局限性，主要表现在：
  - 如果先输入名字，就无法按姓氏进行排序。
  - 如果先输入姓氏，就无法按名字在前姓氏在后（英语国家）的习惯顺序来显示它们。
  - 很难对姓名进行查找。比如说，如果想查找某个姓氏，就必须使用一种模式（pattern）来查找与之匹配的姓名。与直接查找姓氏的做法相比，这种做法在效率和速度方面都存在着不足。

为了避开这些限制，president数据表将把总统们的姓氏和名字分别保存在不同的数据列里。这里把总统们的中间名或有关缩写也安排在名字数据列里。因为我们不太可能对中间名也不太可能对名字进行排序，所以这应该不会影响到对总统姓名进行的排序。这也不会影响到姓名的显示，因为无论是按“Bush, George W.”还是按“George W. Bush”的格式来显示，姓名里的中间名总是紧跟在名字的后面。

还有一个问题需要考虑，有位总统（Jimmy Carter）在他的姓名后面还有一个“Jr.”，应该把它放到哪里呢？根据英语习惯，这位总统的名字既可以写成“James E. Carter, Jr.”，也可以写成“Carter, James E., Jr.”。这个“Jr.”只能出现在整个姓名的末尾，无法与名字或姓氏结合在一起。因此，我们决定再另外创建一个数据列来保存这种姓名后缀。这种情况请大家务必要注意：即使只有一个特例取值，也会影响到数据表示形式的选择。它同时还证明了这样一个事实：应该事先对将要保存到数据库里去的数据的类型做尽可能深入的了解。如果没有事先对这些问题做周密的考虑，就可能会遇到在数据库启用之后还不得不再去修改数据库结构的事情。这种事情虽说不是什么灾难，但还是从一开始就尽量避免它们为好。

- **出生地（州和城市）**——与姓名的情况类似，这些信息也是既可以保存在一个数据列也可以保存在多个数据列。把它们保存在同一个数据列的做法要简单些，但把它们分别保存在不同的数据列将使某些工作更容易完成。比如说，如果把州名与城市名分开放置，诸如“出生在某个州的总统有多少”之类的问题就更容易查询出来。
- **出生日期和逝世日期**——这里需要考虑的特例情况是：不能要求必须填上逝世日期，因为有些总统还依然健在。MySQL有一个专用的特殊值NULL来处理这种“无数据”的情况，



所以我们将逝世日期列里用这个值来表示“依然健在”的情况。

## (2) member数据表

从每条记录都保存着某个人的个人资料的角度看,用来存放“美国历史研究会”会员们的个人资料(member数据表与刚才介绍的president数据表差不多。但member数据表里还包含着其他一些数据列。这个数据表的构成是:

- **姓名**——我们将沿用与president数据表相同的三数据列表示法,即用姓氏、名字、姓名后缀三个数据列来表示会员的姓名。
- **ID编号**——每个会员都会在其会员资格初次生效时分配到一个独一无二的编号。研究会以前从没对会员进行过编号,但既然打算从现在起对会员进行更系统化的管理,那么眼下正是一个好时机。(我希望大家会不断发现MySQL的好处并琢磨出会员资料更多的用途。与会员姓名相比,当需要把member数据表和其他与会员有关的数据表进行关联时,会员编号用起来要简便得多。)
- **失效日期**——会员必须定期续费才能保证其会员资格不会过期失效。在别的项目里,可能需要把这个日期表示为“最近一次交费时的日期”,但这对“美国历史研究会”的情况不适用。会员资格的有效期是一个可变的数字(可以是一年、两年、三年或者五年),而“最近一次交费时的日期”并不能告诉你某个会员必须在何时交纳他的下一期会费。此外,研究会还有一个终身会员制度——虽然可以用一个遥远的未来日期来代表这种情况,但特殊的NULL值更理想,因为用“无数据”来代表“永不失效”是非常合乎逻辑的。
- **电子邮件地址**——电子邮件地址将使兴趣相同的会员彼此更容易进行交流。对于身为研究会秘书的你来说,这些地址将使你能够以电子方式向会员发出续费通知而不必再依靠普通信件。与不得不跑到邮局去寄信的情况相比,这种做法既方便又省钱。还可以利用电子邮件把会员们的个人资料发送给他们,让他们自己去做必要的修改。
- **邮政地址**——这是为那些无法通过电子邮件进行联络(或者没有回复你电子邮件)的会员而准备的。这里将把街道地址、城市名、州名和邮政编码分别保存在不同的数据列里。这里假设全体会员都居住在美国。当然,对于那些在世界各地都有会员的组织机构来说,这个假设过于简单了。如果涉及到来自多个国家的地址,就必须对不同国家所使用的各种地址格式进行研究。比如说,邮政编码并不是一项国际标准,还有些国家被划分为省而不是州。
- **电话号码**——这些信息与地址的作用相似,都是用于与会员进行联络。
- **会员兴趣关键字**——研究会的每位会员都对美国历史感兴趣,但他们的兴趣却可能集中在某些特定的历史时期上。这个数据列就是用来记录这种特殊兴趣的。会员可以利用这些信息来寻找与自己兴趣相同的其他会员。

## (3) 创建“美国历史研究会”数据表

下面,将开始创建“美国历史研究会”的各种数据表。我们使用CREATE TABLE语句来完成这一工作,这条语句的格式是:

```
CREATE TABLE tbl_name ( column_specs );
```

其中, *tbl\_name*是你给数据表起的名字; *column\_specs*则是该数据表里的各个数据列以及各种索引(如果有的话)的定义。索引能够加快信息的检索速度,将在第4章中进行介绍。下面是对应于president数据表的CREATE TABLE语句:

```
CREATE TABLE president
(
    last_name    VARCHAR(15) NOT NULL,
    first_name   VARCHAR(15) NOT NULL,
    suffix       VARCHAR(5) NULL,
    city         VARCHAR(20) NOT NULL,
    state        VARCHAR(2) NOT NULL,
    birth        DATE NOT NULL,
    death        DATE NULL
);
```

如果打算亲自输入这条语句,请先用下面的命令来调用mysql客户程序并把数据库sampdb设置为当前的默认数据库:

```
% mysql sampdb
```

然后再敲入上面的CREATE TABLE语句。不要漏掉语句末尾的分号,这样才能让mysql知道这条语句的结束位置。

如果打算利用一个预先写好的脚本文件来创建president数据表,可以使用sampdb发行版本里的create\_president.sql文件。可以在系统安装这个发行版本时所创建的sampdb目录里找到这个文件。先切换到这个目录,然后再执行下面这条命令:

```
% mysql sampdb < create_president.sql
```

不管如何调用mysql客户程序,请都不要忘记在命令行里的数据库名称之前加上必要的连接参数(主机名、用户名、口令等)。

CREATE TABLE语句中的数据列定义由以下几部分组成:数据列的名字、该数据列的数据类型(这个数据列是用来保存哪种数据的)、该数据列的属性。

president数据表用到了两种列类型(column type): VARCHAR和DATE。VARCHAR(*n*)的意思是:这个数据列里存放着长度可变的字符(字符串)值,其最大长度是*n*个字符。这个*n*要根据对字符串数据长度的预估来选定。比如说,把state数据列定义为VARCHAR(2)类型,而这是根据美国州名都可以缩写为两个字母而确定的。其他字符串类型的数据列所要容纳的值可能会比较长,所以它们的宽度也要大一些。

我们用到的另一种列类型是DATE。毋庸置疑,这种类型的数据列是用来保存日期值的,但大家千万要注意日期值的表示格式。MySQL要求日期表示为'CCYY-MM-DD'的格式,其中,CC、YY、MM、DD分别代表世纪、年份、月份和日期。这是ANSI SQL中规定的日期表示标准(也叫做ISO 8601格式)。比如说,2002年7月18日在MySQL里必须被表示为'2002-07-18',而不是'07-18-2002'或'18-07-2002'。

president数据表还用到了两种数据列属性:NULL(表示“无数据”)和NOT NULL(表示

“不得为空”)。表中的大部分数据列都具有NOT NULL属性——因为必须要在其中填上数据。具有NULL属性的数据列有两个,一个是suffix(姓名后缀,大多数总统的姓名里都没有后缀),另一个是death(逝世日期,有些总统还活着,用不着填逝世日期)。

下面是用来创建member数据表的CREATE TABLE语句:

```
CREATE TABLE member
(
    member_id    INT UNSIGNED NOT NULL AUTO_INCREMENT,
    PRIMARY KEY (member_id),
    last_name    VARCHAR(20) NOT NULL,
    first_name   VARCHAR(20) NOT NULL,
    suffix       VARCHAR(5) NULL,
    expiration   DATE NULL DEFAULT '0000-00-00',
    email        VARCHAR(100) NULL,
    street       VARCHAR(50) NULL,
    city         VARCHAR(50) NULL,
    state        VARCHAR(2) NULL,
    zip          VARCHAR(10) NULL,
    phone        VARCHAR(20) NULL,
    interests    VARCHAR(255) NULL
);
```

可以亲自输入这条语句,也可以利用sampdb发行版本里的脚本文件并执行下面的命令:

```
% mysql sampdb < create_member.sql
```

在member数据表里,大部分数据列的类型都是可变长度的字符串,只有两个数据列(用来保存会员号的member\_id和用来保存失效日期的expiration)是例外。

为了避免出现把不同的会员当做同一个人的混乱局面,数据列member\_id必须满足这样一个要求:该数据列里的值必须是独一无二的。这正是MySQL里的AUTO\_INCREMENT数据列大显身手的地方——当向member数据表里添加新记录时,MySQL能在member\_id列自动生成一个独一无二的会员号。虽然member\_id数据列里的数据只是些数字,但它的声明却包含了好几个部分:

- INT——表示这个数据列将用来保存整数值(没有小数部分的数字)。
- UNSIGNED——不允许出现负数。
- NOT NULL——必须填有数据,不得为空。(这意味着每个会员都有一个会员号,不可能出现没有会员号的会员。)
- AUTO\_INCREMENT——这是MySQL里的一个特殊属性。它表示有关的数据列里存放的是序列编号。AUTO\_INCREMENT机制的工作原理是这样的:当向member数据表里插入数据记录时,如果没有给出member\_id列的值(或者给出的值是NULL),MySQL将自动生成序列中的下一个编号并把该编号赋值给这个数据列。这样,为新会员分配会员号的工作就简单了,因为MySQL可以自动完成。

PRIMARY KEY子句表示需要对member\_id数据列进行索引以加快查找速度,同时也要求该数据列里的各个值都必须是独一无二的——这正好满足了对会员ID的编号要求,因为我们绝不



希望出现误把同一个会员号分配给两个会员的情况。(此外,MySQL本身也要求每一个具备AUTO\_INCREMENT属性的数据列必须拥有某种形式的惟一索引,换句话说,如果我们忘了给member\_id数据列加上一个惟一索引,member数据表的定义就不合法。)

如果读者还不能理解AUTO\_INCREMENT和PRIMARY KEY的含义与作用,不妨把它想像成一种用来为每位会员生成一个独一无二的会员号的魔术好了。我们真正关心的是那些会员号是否都是独一无二的,它们到底等于多少并不重要。(有关AUTO\_INCREMENT数据列在声明与应用方面的详细讨论见第2章。)

数据列expiration是一个DATE(日期)类型的数据列,它有一个默认值'0000-00-00',用这个值来表示“尚未输入一个合法数据”。之所以不把这个默认值设置为NULL,是因为我们还需要用NULL来代表“某会员是一位终身会员”的情况。如果没有专门设定一个默认值,那么,当我们为某个新会员往member数据表里插入一条新记录时,万一忘记了设定其会员资格的失效日期,MySQL就会自动地在expiration数据列里填上一个表示“无数据”的NULL值,使这名新会员莫名其妙地成为一名终身会员!把expiration数据列的默认值设定为'0000-00-00',不仅能避免这种问题,而且使我们能够利用这个默认值定期地把那些失效日期输入有误的会员记录给查找出来。

现在,MySQL已经创建了两个数据表,下面检查一下它做得怎么样。在mysql里,可以用下面这条命令来查看president表的结构:

```
mysql> DESCRIBE president;
```

Field	Type	Null	Key	Default	Extra
last_name	varchar(15)				
first_name	varchar(15)				
suffix	varchar(5)	YES		NULL	
city	varchar(20)				
state	char(2)				
birth	date			0000-00-00	
death	date	YES		NULL	

在MySQL的某些版本里,DESCRIBE命令的执行结果里还包括有一些诸如访问权限之类的附加信息。出于突出重点和简洁方面的考虑,这里把那些附加信息都省略了。

上面这条DESCRIBE命令的输出结果基本符合我们的预期,只是state数据列的类型由当初定义的VARCHAR(2)变成了CHAR(2)。这是怎么回事?MySQL为什么会把state数据列的类型由VARCHAR(2)“偷偷地”改成CHAR(2)呢?这与短字符串数据列的存储效率有关,这里不深入讨论这个问题,有兴趣的读者可以去参阅第3章中关于ALTER TABLE语句的讨论。从就事论事的角度看,这两种类型并没有什么区别——它们都表示该数据列将用来存放一个由两个字符构成的值。

如果发出的是DESCRIBE member命令,mysql就将显示有关member数据表的类似信息。

DESCRIBE是一个非常有用的命令,尤其是当你只记得数据表的名字却想不起其中某个数据列的名字、类型或数据宽度等细节的时候。还可以利用这条命令来查看各数据列在数据行里

存储的先后顺序，这个顺序很重要——INSERT或LOAD DATA等语句要求各数据列的值必须按它们默认的存储顺序依次列出。

能够用DESCRIBE命令查出来的信息也可以通过别的手段获得。可以把它简写为DESC，也可以把它写成EXPLAIN或SHOW语句。下面这些语句的作用是相同的：

```
DESCRIBE president;
DESC president;
EXPLAIN president;
SHOW COLUMNS FROM president;
SHOW FIELDS FROM president;
```

这些语句还允许你把输出内容限制为某几个特定的数据列。比如说，如果在SHOW语句的末尾加上一个LIKE子句，就只能看到与给定模式相匹配的那几个数据列的有关信息，如下所示：

```
mysql> SHOW COLUMNS FROM president LIKE '%name%';
+-----+-----+-----+-----+-----+-----+
| Field      | Type          | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| last_name  | varchar(15)   |      |     |          |       |
| first_name | varchar(15)   |      |     |          |       |
+-----+-----+-----+-----+-----+-----+
```

这里使用的百分号(%)是一个特殊的通配符，将在稍后的“模式匹配”小节里对通配符进行介绍。也可以给DESCRIBE和EXPLAIN语句加上类似的限制条件，附录D对有关语法做了详细的讨论。

SHOW语句还有其他几种用法，可以用来从MySQL获取各种信息。SHOW TABLES能够把当前数据库里的数据表列出来。比如说，我们已经在sampdb数据库里创建了两个数据表，该语句的输出结果为：

```
mysql> SHOW TABLES;
+-----+
| Tables_in_sampdb |
+-----+
| member           |
| president        |
+-----+
```

SHOW DATABASES能够把当前连接的服务器上的数据库列出来，如下所示：

```
mysql> SHOW DATABASES;
+-----+
| Database |
+-----+
| menagerie |
| mysql     |
| sampdb    |
| test      |
+-----+
```

服务器不同，这条语句列出来的数据库清单也就不同，但至少应该能看到sampdb和mysql。sampdb数据库是我们刚创建的，而名为mysql的数据库则存放着各种用来控制MySQL访问权限的权限分配表。

工具程序mysqlshow是与SHOW语句功能相当的命令行接口，用SHOW语句能查看到的信息也都能用mysqlshow程序查看到。不带参数的mysqlshow程序将列出一份数据库清单：

```
% mysqlshow
+-----+
| Databases |
+-----+
| menagerie |
| mysql     |
| sampdb    |
| test      |
+-----+
```

如果给它加上一个数据库名，mysqlshow将列出一份给定数据库里的数据表清单：

```
% mysqlshow sampdb
Database: sampdb
+-----+
| Tables |
+-----+
| member |
| president |
+-----+
```

如果同时给出数据库名和数据表名，mysqlshow将把那个数据表里各数据列的有关信息列出来——就像SHOW COLUMNS语句那样。

## 2. “考试记分项目”的数据表

要想确定“考试记分项目”都需要用到哪些数据表，先要弄清楚你是怎样用纸质记分簿来记录考生成绩的。请看图1-2，假设这是纸质记分簿里的某一页，上面是一个记有考试分数的表格。除考试分数外，这个表格还有一些使考试分数变得有实际意义的信息。学生的姓名和ID号列在表格的左侧（为简洁起见，这里只列出了4名学生）；考试或测验的举行日期则列在了表格的顶部。根据这份表格，可以知道在9月3日、6日、16日、23日对学生进行了测验，在9月9日和10月1日对学生进行了考试。

学生		分数						
ID	姓名	Q 9/3	Q 9/6	T 9/9	Q 9/16	Q 9/23	T 10/1	...
1	Billy	14	10	73	14	15	67	...
2	Missy	17	10	68	17	14	73	...
3	Johnny	15	10	78	12	17	82	...
4	Jenny	14	13	85	13	19	79	...
...	...	...	...	...	...	...	...	...

图1-2 纸质记分簿里的某一页



要想把这些信息记录到数据库里，就需要有一个score数据表。那么，这个数据表里的各条记录都应该包含有哪些信息呢？这个问题不难回答。在每个数据行里，需要列出学生的姓名（name）、考试或测验的日期（date）和学生的考试分数（score）。图1-3给出了纸质记分簿里的一些考试分数被表示在数据表里的样子。（注意，scores数据表里的日期是按MySQL的日期表示法'CCYY-MM-DD'格式写出来的。）

score 数据表

name	date	score
Billy	2002-09-23	15
Missy	2002-09-23	14
Johnny	2002-09-23	17
Jenny	2002-09-23	19
Billy	2002-10-01	67
Missy	2002-10-01	73
Johnny	2002-10-01	82
Jenny	2002-10-01	79

图1-3 最初的score数据表

可是，如此得到的数据表是有问题的，它丢失了某些信息。比如说，仔细看看图1-3就会发现，我们无法根据这些分数来区别考试与测验。一般说来，在评定学生们的期末总成绩时，考试分数与测验分数还是有一定区别的，所以我们有必要知道这些考分的类型。当然，可以根据某给定日期的分数范围（测验分数通常要比考试分数在数值上低很多）来推测出这个类型，但这种不用数据明确表明而纯粹依靠推测的做法却未免有点蠢笨。

要想在每条记录里把考分类型区别开还是有办法的，比如说，可以像图1-4那样给score数据表增加一个数据列，并用T或Q来分别代表test（考试）或quiz（测验）。这种做法的好处是考分类型能直接体现在数据上，但让人不满意的是这部分信息有些冗余。看看那些日期相同的记录就能发现，考分类型（type）列里的值全都是一模一样的：在9月23日，所有考分的类型全都为Q；而在10月1日，所有考分的类型又全都是T。这有点太啰嗦了。要是按这种办法来记录学生们的考试分数，不仅要反复输入一个相同的日期，而且还要反复输入一个相同的考分类型。有谁愿意反复输入这么多的冗余信息呢？

score 数据表

name	date	score	type
Billy	2002-09-23	15	Q
Missy	2002-09-23	14	Q
Johnny	2002-09-23	17	Q
Jenny	2002-09-23	19	Q
Billy	2002-10-01	67	T
Missy	2002-10-01	73	T
Johnny	2002-10-01	82	T
Jenny	2002-10-01	79	T

图1-4 修改后的score数据表，增加了一个type（考分类型）列

我们应该想出一个更好的办法。与其把考分类型放到score数据表里，不如把它与考试日期对应起来。可以把考试日期列成一个表，再把各日期里发生的“考试事件”（测验或考试）记录

在这个表里。这样，只要根据score表里的日期在event表里查出当天的考试事件类型，就能知道某个分数是来自一次测验还是来自一次考试。图1-5给出了这种思路下的数据表布局，并以2002年9月23日为例画出了score表与event表的对应关系：根据score数据表里的date（'2002-09-23'），从event数据表里查出当天举行的是一次测验，所以score表里的那个分数是一次测验成绩。

score 数据表			event 数据表	
name	date	score	date	type
Billy	2002-09-23	15	2002-09-03	Q
Missy	2002-09-23	14	2002-09-06	Q
Johnny	2002-09-23	17	2002-09-09	T
Jenny	2002-09-23	19	2002-09-16	Q
Billy	2002-10-01	67	2002-09-23	Q
Missy	2002-10-01	73	2002-10-01	T
Johnny	2002-10-01	82		
Jenny	2002-10-01	79		

图1-5 通过date列相关联的score与event数据表

与通过推测来判断考分类型的做法相比，新办法要好得多——现在可以从显式地记录在数据库里的数据直接推导出考试分数的类型。与把考分类型直接记录在score数据表里的做法相比，新办法也要好得多——只需记录一次考分类型，不必再为每个考分都记录一次了。

话又说回来，现在需要把多个数据表的信息结合起来才行。如果你和我一样，在第一次听说这种事的时候，你可能会想：“嘿，这个主意可真够酷的。可这么多的数据表，想查什么东西会不会太费事？这会不会把事情搞得更复杂呢？”

从某种意义上讲，这种担心是有道理的。两个表当然要比一个表更复杂。可仔细看看当初的记分簿（如图1-2所示），你不是已经在记录两套信息了吗？请大家注意以下这两个事实：

- 把学生们的考试分数记录在表格的每一小格里，这些小格按学生姓名和考试日期排列（按姓名，由上往下排列；按日期，由左往右排列）。这正是上面所说的两套信息中的一套，与这套信息相对应的是score数据表里的内容。
- 如何知道各日期所代表的事件类型呢？你在记分簿里是这样做的：在日期的上面写上一个T或Q——在表格的顶部把考试日期与考试类型关联起来。这正是上面所说的两套信息中的第二套，与这套信息相对应的是event数据表里的内容。

换句话说，也许你本人还没有意识到这一点，但你在记分簿里做的事与把信息放到两个数据表里的情况并没有什么差异，即便是有差异，也只是纸质记分簿里的两套信息没有明确地分别放置而已。

记分簿表格的例子体现出了人们对信息的某些想法，也反映出这样一个问题：把信息妥善地放到数据库里去并不是一件简单的事情。在日常生活中，人们习惯于把不同信息综合起来并作为一个整体来考虑。但数据库毕竟不是我们人类的大脑，而这正是它们看起来过于人工化和不太自然的原因之一。人类习惯于把信息综合在一起的思维特点使我们有时很难意识到自己正使用着多种类型的信息而非孤零零的一种。因此，如何以“想数据库所想”的方式来表达信息数据成为一个很富有挑战性的过程。



图1-5里的event数据表还隐含地体现了这样一个要求：date列里的日期必须是独一无二的，因为每一个日期都将用来关联score和event数据表里的某些数据记录。换句话说，不得在同一天进行两场测验（或者一场测验加一场考试）。如果这样做了，那么，score数据表里将会有两组考分记录、event数据表里将会有两条记录，它们都对应于同一个日期，而这意味着通过日期的匹配关系来关联score记录和event记录的做法将难以为继。

假如每天最多进行一场考试，那么这个问题就不成为问题。但一天两场考试的情况真的是永远都不会发生吗？也许如此——心地善良的你应该不会对学生们苛刻到要对他们一天进行一场考试和一场测验的程度。不过（希望大家别怪我多嘴），我经常听见有些人说“这种奇怪的事情永远也不会发生”，可奇怪的事情却真的在某个时刻发生了；而为了弥补这一漏洞，这些人就不得不加班加点地去重新设计他们的数据表。

与其临时抱佛脚，不如防患于未然，提前预见到可能出现的各种问题并准备好应对措施将会减少很多麻烦。因此，我认为还是现在就对“会在同一天记录两组考试分数”的情况做一下分析比较好。应该如何解决这个问题呢？别担心，随着讨论的深入，将发现这个问题并没有看上去那么困难。只要对有关数据的布局结构做一个小小的改动，在同一天发生多次考试事件的事情就不再会引起麻烦：

1) 在event表里增加一个数据列，利用它给event表里的各个记录分配一个独一无二的编号。实际上，这等于是给各次考试事件分别赋予了一个独一无二的ID编号，因此命名这个新增的数据列为event\_id。（虽然看着有点奇怪，可这一做法却并不是什么新点子，图1-2中的记分簿表格其实已经用到了这个特性——记分簿表格分数记录部分的列编号就相当于这里所说的事件编号。虽说没有把各列的编号明确地写出来并标明是“event ID”，但它的的确确是存在的。）

2) 在把考试分数记到score数据表里的时候，用考试事件的ID编号来代替考试日期。

完成上述改动后，得到了如图1-6所示的结果。现在，score和event表必须用event\_id（事件编号）而不是date（日期）来进行关联，使用event表不仅能查出考分的类型，而且还能查出它具体发生在哪一天。最重要的是，event表中必须具备惟一性的不再是日期，而是事件编号。这意味着即使在一天之内进行了十几场考试，也能条理清晰地把各场考试的分数全都记录下来。（毫无疑问，学生们肯定害怕听到这个消息。）

score 数据表

name	event_id	score
Billy	5	15
Missy	5	14
Johnny	5	17
Jenny	5	19
Billy	6	67
Missy	6	73
Johnny	6	82
Jenny	6	79

event 数据表

event_id	date	type
1	2002-09-03	Q
2	2002-09-06	Q
3	2002-09-09	T
4	2002-09-16	Q
5	2002-09-23	Q
6	2002-10-01	T

图1-6 通过事件编号列相关联的score与event数据表

然而，从人类的角度看，图1-6里的表格不如前面那几个看起来顺眼。score表变得越来越抽象，数据列的含义也越来越不容易看懂。请看图1-4里的score表，其中既有考试日期又有考分类型，让人一眼就能看出这个表是干什么用的。但在图1-6里，这两个数据列却不见了，所看到的是一个高度抽象化的信息表示形式。如果没有以上这些解说，谁能一眼看出score表的event\_id到底是什么意思？谁又会喜欢查一个这样的表格？

此时此刻，我们来到了一个十字路口。此前，大家对电子化的考试记分系统充满希望，觉得很快就能从繁琐的评分工作中解脱出来，但在看过上面的讨论后，却发现单是把信息放到数据库里就已经很不容易做到最好了。高度抽象的信息与它们所代表的事物似乎毫无联系，这往往会让人们产生一种畏难情绪。

这很自然地引出了一个问题：“干脆不用数据库会不会更好？也许MySQL不适合我。”我的回答大家肯定都能猜到，要不这本书就不会有这么厚了。但对读者们来说，在项目开工前多考虑几种办法总是好的。应该问自己：是使用MySQL这样的数据库系统好，还是使用电子表格程序等其他办法好？从一方面看：

- 记分簿由行和列构成，电子表格也是如此。它们二者在概念和直观上都很相似。
- 电子表格程序能够进行计算，所以分数统计工作不难完成。测验分数和考试分数可能不太容易按不同权重来统计，但肯定有办法解决。

另一方面，如果只想对部分数据进行操作（比如，只统计测验分数或者只统计考试分数），或者想进行某种对比分析（比如男生与女生的成绩对比），或者想灵活地汇总和显示各种统计信息，事情就不同了。这些工作电子表格都不擅长，关系数据库系统则能大显身手。

往开处想，信息的高度抽象化也不是什么大不了的事。只是需要在数据库建立之初考虑信息在数据库里的表示方式，按照最符合目标的要求来设置它们。在数据库建立起来以后，信息数据的提取和显示工作将由数据库引擎按一定的逻辑来完成，所看到的是有意义的资料，而不是抽象的信息零件。

比如说，从score数据表检索学生分数的时候，想看到考试日期而不是事件编号。这很容易办到：数据库将根据事件编号从event表里查出考试日期来。如果还想知道考试分数是来自一次测验还是来自一次考试，这也很容易办到：数据库也能查出考分类型——仍要以事件编号为根据。别忘了，MySQL之类的关系数据库系统最擅长的本领是把一样东西与另一样东西关联起来，从多个信息源把你最想知道的信息给查找出来。在考试记分的例子里，MySQL必须通过事件编号才能对信息进行关联和提取，而你（数据库的使用者）并不需要关心这类细节。

为了让大家对MySQL的信息关联和信息提取机制有一个了解，我们准备了一个例子，假定你打算查看2002年9月23日的考试分数。下面这个查询将把给定日期里的考试分数查出来：

```
SELECT score.name, event.date, score.score, event.type
FROM score, event
WHERE event.date = '2002-09-23'
AND score.event_id = event.event_id;
```

长得有点吓人，对吗？这个查询将把score表和event表结合（关联）起来并检索出学生姓名、考试日期、考试分数和考分类型等信息，下面是它的输出结果：



name	date	score	type
Billy	2002-09-23	15	Q
Missy	2002-09-23	14	Q
Johnny	2002-09-23	17	Q
Jenny	2002-09-23	19	Q

首先，是不是觉得上面这个表格有点面熟？没错，它与图1-4里的表格格式是一模一样的。其次，不必知道事件编号就能得到这份查询结果。你指定了一个日期，MySQL把该日期里的考试分数找了出来。总之，虽然数据库里的信息很抽象，与它们所代表的事物似乎也没有直观的联系，但并不会影响到它们的使用，数据库会根据查询把信息提取出来并显示为有意义的资料。

再仔细看看这个查询，大家也许又会产生一些新的问题。别的不说，这个查询看起来也太长太复杂了点，仅为查出某日期的考试分数就要写这么多东西是不是太复杂了？是的。但是，有好几种办法能避免在输入查询命令的时候敲很多行内容。比较常见的做法是：一旦确定了某个查询的最终写法，就把它保存起来，以后在必要时就可以直接使用它了。将在第1.5节里对这方面内容做进一步讨论。

要不是为了让大家对查询过程有个了解，我是不想这么早就给出例子来的。事实上，与我们真正用来检索考试分数的查询相比，刚才举的例子还算是简单的。为什么这么说呢？因为我们还需要对有关数据表的布局再做一次较大的改动。首先，让score表不再包含学生姓名，我们将使用一个独一无二的学生ID编号，这其实就等于是用纸质记分簿里的ID列而不是Name列来构成score表。再新建一个名为student的数据表来存放学生姓名（name）和学号（student\_id）（如图1-7所示）。



图1-7 通过学生学号和事件编号相关联的student、score、event数据表

为什么要做这样的改动呢？这是为了预防出现两名学生名字相同的情况，独一无二的学生ID将有助于把他们区分开来。（这与我们不使用日期而使用独一无二的事件编号来区分同一天进行的多场考试的分数的道理是一样的。）在对数据表的布局做了上述改动之后，用来查询给定日期的考试分数的命令又变得稍微复杂了一些，如下所示：

```
SELECT student.name, event.date, score.score, event.type
FROM event, score, student
WHERE event.date = '2002-09-23'
AND event.event_id = score.event_id
AND score.student_id = student.student_id;
```

如果你现在还看不懂这个查询命令，请不要着急，大多数初学者都是如此。在这个教程的后半部分内容里，还会遇到这个查询命令，到那时你就能看明白它了。真的，不是开玩笑。

大家可能已经注意到，在图1-7的student表里增加了一些记分簿里没有的东西，它多了一个sex（性别）列。可以利用这个数据列来统计班级里男生或女生的人数，也可以利用它来对男女生的成绩进行对比分析。

考试记分项目的数据表到这里已经设计得差不多了，只需再增加一个用来记录考试缺勤情况的数据表就全部完成了。这个数据表的内容很简单：一个学生ID和一个日期（如图1-8所示）。这个数据表里的每一个数据行都代表着考试当天缺勤的一名学生。等到学期结束的时候，将通过MySQL的统计功能来汇总这个表里的数据，把每名学生的缺勤次数查出来。

absence 数据

student_id	date
2	2002-09-02
4	2002-09-15
2	2002-09-20

图1-8 absence数据表

好了，考试记分项目的数据表到这里就全部设计完成了，下一步就该把它们创建出来了。下面是用来创建student数据表的CREATE TABLE语句：

```
CREATE TABLE student
(
    name          VARCHAR(20) NOT NULL,
    sex           ENUM('F','M') NOT NULL,
    student_id    INT UNSIGNED NOT NULL AUTO_INCREMENT,
    PRIMARY KEY (student_id)
);
```

可以在mysql客户程序里敲入上述语句，也可以在命令行上执行如下所示的命令：

```
% mysql sampdb < create_student.sql
```

上面这条CREATE TABLE语句将创建一个名为student且包含有name、sex、student\_id三个数据列的数据表。

name是一个可变长度的字符串数据列，它最多可以容纳20个字符。这种人名表示法要比“美国历史研究会”场景的数据表里使用多个数据列来分别保存人的姓氏和名字的情况简单，它只用了一个数据列。之所以这样做，是因为考试记分项目不会出现必须用多个数据列来表示人名的查询操作。

sex用来表明某学生是男生还是女生。这是一个ENUM（枚举）类型的数据列，这种数据列的可取值只能是在该数据列的定义里枚举出来的那些值中的某一个：'F'代表女生，'M'代表男生。如果想把某个数据列的可取值限制在一个元素个数有限的集合内，使用ENUM非常合适。当然，也可以把这个数据列定义为CHAR(1)，但ENUM能够更明确地把这个数据列只有有限个取值的特点表示出来。如果你忘了它都有哪些可能的取值，可以发出一条DESCRIBE tbl\_name命令来查看。对于ENUM数据列，MySQL将把它合法的枚举值都列出来。

顺便说一下，ENUM数据列的值不一定是单个字符。可以把这个数据列定义为：ENUM('female','male')。

student\_id是一个整数类型的数据列，用它来保存独一无二的学生ID编号。一般说来，学生

们的学号应该从一个权威机构（比如学校办公室）来获得，但既然这只是一个示例性的数据表，就不妨自己编造一些，我们使用了一个AUTO\_INCREMENT数据列，对这种数据列的介绍请参见前面创建member数据表时对member\_id数据列的讨论。

需要提醒大家的是，如果真的是从学校办公室获得的学生ID编号而不是自动生成这些编号的，就千万不要在定义student\_id数据列的时候给它加上AUTO\_INCREMENT属性；但为了避免学生们的ID编号出现重复，PRIMARY KEY子句还是要保留下来的。

下面是用来创建event数据表的CREATE TABLE语句：

```
CREATE TABLE event
(
    date          DATE NOT NULL,
    type          ENUM('T','Q') NOT NULL,
    event_id      INT UNSIGNED NOT NULL AUTO_INCREMENT,
    PRIMARY KEY (event_id)
);
```

为了创建这个数据表，既可以在mysql客户程序里敲入上述语句，也可以在命令行上执行如下所示的命令：

```
% mysql sampdb < create_event.sql
```

所有数据列都被定义为NOT NULL，因为它们的内容都不允许为空。

date数据列将用来保存标准的MySQL日期值，即必须写成'CCYY-MM-DD'的格式。

type代表着考试分数的类型。像student表里的sex列一样，type也是一个枚举类型的数据列。它的可取值是'T'和'Q'，分别代表test（考试）和quiz（测验）。

event\_id是一个AUTO\_INCREMENT类型的数据列，并同时被声明为PRIMARY KEY，它与student数据表里的student\_id数据列情况类似。利用AUTO\_INCREMENT属性，就能方便地生成独一无二的事件编号了。与student数据表里的student\_id数据列类似，这些编号到底是多少并不重要，重要的是它们必须是独一无二的。

下面是用来创建score数据表的CREATE TABLE语句：

```
CREATE TABLE score
(
    student_id    INT UNSIGNED NOT NULL,
    event_id      INT UNSIGNED NOT NULL,
    PRIMARY KEY (event_id, student_id),
    score         INT NOT NULL
);
```

为了创建这个数据表，既可以在mysql客户程序里敲入上述语句，也可以在命令行上执行如下所示的命令：

```
% mysql sampdb < create_score.sql
```

student\_id和event\_id数据列都是INT（整数）类型的数据列，它们分别代表着每一个考试分



数所对应的学生和考试事件。通过它们把student和event数据表关联起来，可以查出学生姓名和考试日期。我们还把这两个数据列的组合定义为PRIMARY KEY，这将确保数据表里不会重复出现考试事件与学生编号全都相同的两条记录，另外也为今后修改某个考试分数的操作提供了便利——如果把某个考试分数输错了，就可以用MySQL的REPLACE语句重新输入一条新记录来覆盖掉出错的老记录。用不着先DELETE（删除）再INSERT（插入）那么费事，MySQL会自动完成。

注意，把student\_id和event\_id的组合定义为PRIMARY KEY的含义是：只要求它们的组合必须是独一无二的。在score表里，这两个数据列里的值本身都不是独一无二的，相同的event\_id会出现在多个score记录里（对应于每位学生），相同的student\_id也会出现在多个score记录里（对应于每次考试或测验）。

score数据列是一个整数类型的数据列——假设考试分数永远是整数。如果想用到58.5这样的小数类型考试分数，就得把这个数据列定义为一个浮点类型（如FLOAT或DECIMAL等）的数据列。

下面是用来创建absence数据表的CREATE TABLE语句：

```
CREATE TABLE absence
(
    student_id INT UNSIGNED NOT NULL,
    date       DATE NOT NULL,
    PRIMARY KEY (student_id, date)
);
```

为了创建这个数据表，既可以在mysql客户程序里敲入上述语句，也可以在命令行上执行如下所示的命令：

```
% mysql sampdb < create_absence.sql
```

student\_id和date数据列都被声明为NOT NULL，因为它们的内容都不允许缺失。为了避免因工作失误而使这个数据表里出现内容重复的记录项，把这两个数据列的组合也定义为PRIMARY KEY——不管怎么说，把学生缺勤一天的情况统计两次肯定是不公平的！

#### 1.4.7 添加新记录

现在，已经把数据库和其中的数据表都创建出来了。接下来，需要往数据表里放一些记录项。但在此之前，先介绍一下如何对数据表里的内容进行查找——在往里面放了一些记录之后，总应该先看看自己做得怎么样吧。虽然把有关检索操作的详细介绍安排在下一小节里，但读者至少应该先把下面这条语句弄明白，它是用来查看名为tbl\_name的数据表里的内容的：

```
SELECT * FROM tbl_name;
```

比如说：

```
mysql> SELECT * FROM student;
Empty set (0.00 sec)
```

现在,mysql会报告说这个数据表是空的,但经过本小节中的几次示例操作之后,就会看到不同的结果了。

往数据库添加数据的办法有好几种。可以用INSERT语句以手工方式逐条地把记录插入到数据表里;也可以利用一个文件把记录添加到数据表里,这个文件的内容既可以是一系列提前输入好的INSERT语句(数据将通过客户程序mysql加载到数据库里去),也可以是原始数据值(将通过LOAD DATA语句或mysqlimport工具程序加载到数据库里去)。

本小节将对把记录插入到数据表里去的各种办法进行介绍。大家应该对它们都进行练习,熟悉并掌握它们的工作原理和用法。练习完这些办法之后,利用本小节末尾部分给出的命令丢弃(删除)各有关数据表,重新创建它们,再把一整套已知数据加载到里面去。这样,大家数据库里的内容就与将在以后内容的示例中用到的数据保持一致了,而自己做示例练习时看到的结果也将会与书中给出的结果保持一致。(如果你已经知道如何插入数据记录,可以直接跳到本小节末尾去充实自己的数据表。)

下面先来学习如何用INSERT语句添加数据记录。这是一条SQL语句,用它来指定打算往哪个数据表插入一个数据行以及该数据行的各数据列的值。INSERT语句有以下几种形式:

- 可以一次性地列出全部数据列的值,如下所示:

```
INSERT INTO tbl_name VALUES(value1,value2,...);
```

比如说:

```
mysql> INSERT INTO student VALUES('Kyle','M',NULL);
mysql> INSERT INTO event VALUES('2002-9-3','Q',NULL);
```

在使用这个语法的时候,关键字VALUES后面的括号里必须为数据表的全体数据列准备好对应的值,这些值的先后顺序也必须与各数据列在数据表里的存储顺序保持一致。(这个顺序通常是各数据列在用来创建这个数据表的CREATE TABLE语句里的出现顺序。)如果不知道数据列的先后顺序,可以先用一条“DESCRIBE *tbl\_name*”语句来查一下。

MySQL里的字符串或日期值必须放在一对引号里(单引号或双引号都行)才能被引用。NULL值对应于student和event数据表里的AUTO\_INCREMENT数据列。在AUTO\_INCREMENT数据列里插入一个表示“无数据”的NULL值将使MySQL为这个数据列自动生成下一个序号。

MySQL 3.22.5及以后的版本允许用一条INSERT语句把多个数据行插入到数据表里去,具体语法如下:

```
INSERT INTO tbl_name VALUES(...),(...),...;
```

例如:

```
mysql> INSERT INTO student VALUES('Abby','F',NULL),('Kyle','M',NULL);
```

与刚才必须使用多条INSERT语句的情况相比,这种做法不仅能让你少打不少字,而且还能提高服务器的执行效率。

- 可以直接对数据列进行赋值，先给出数据列的名字，再列出它的值。这特别适用于所创建的记录只有少数几个数据列需要有初始值的情况。具体语法如下：

```
INSERT INTO tbl_name (col_name1,col_name2,...) VALUES(value1,value2,...);
```

例如：

```
mysql> INSERT INTO member (last_name,first_name) VALUES('Stein','Waldo');
```

MySQL 3.22.5及以后的版本允许这种形式的INSERT语句一次插入多个记录：

```
mysql> INSERT INTO student (name,sex) VALUES('Abby','F'),('Kyle','M');
```

没有在INSERT语句中出现的数据列将被赋予其默认值。比如，上面两条语句没有给出member\_id或event\_id数据列的值，所以MySQL将把默认值NULL赋给它们。（又因为member\_id和event\_id都是AUTO\_INCREMENT数据列，所以结果将是这两个数据列被赋值为MySQL自动生成的下一个序列号，这与直接把NULL赋值给它们的效果是一样的。）

- 从MySQL 3.22.10开始，还可以用col\_name = value格式对数据列进行赋值。

```
INSERT INTO tbl_name SET col_name1=value1, col_name2=value2, ... ;
```

例如：

```
mysql> INSERT INTO member SET last_name='Stein',first_name='Waldo';
```

没有在SET子句里出现的数据列将被赋予其默认值。这种形式的INSERT语句不允许一次插入多个数据行。

把数据记录加载到数据表里的另一种方法是从一个文件里把它们直接读出来。比如说，如果有一个名为insert\_president.sql的文件，文件中包含用来把新记录添加到president数据表里的INSERT语句，就可以像下面这样直接执行之：

```
% mysql sampdb < insert_president.sql
```

（可以在sampdb发行版本里找到这个insert\_president.sql文件。）如果已经进入mysql，可以用一条SOURCE命令来读入这个文件，如下所示：

```
mysql> SOURCE insert_president.sql;
```

SOURCE命令只能用在MySQL 3.23.9或更高的版本里。

如果文件里的记录项不是以INSERT语句而是以原始数据值的形式来存放的，可以利用LOAD DATA语句或mysqlimport工具程序来加载它们。

LOAD DATA语句就像是一架大型装载机，它能把文件里的数据一次性地全部读到数据表里。这条语句要在mysql客户程序里使用：

```
mysql> LOAD DATA LOCAL INFILE 'member.txt' INTO TABLE member;
```

假设数据文件member.txt就保存在客户主机上的当前目录里，上面这条语句将读这个文件并把它的内容发送到member数据表所在的服务器去加载之。（可以在sampdb发行版本里找到这个member.txt文件。）



在默认情况下, LOAD DATA语句将假设: 1) 各数据列的值以制表符分隔; 2) 各数据行以换行符分隔; 3) 数据值的排列顺序与各数据列在数据表里的先后顺序相一致。但可以用它来读取其他格式的数据文件或者按其他顺序来读取各数据列的值, 有关细节请参阅附录D。

LOAD DATA LOCAL只能用在MySQL 3.22.15或更高的版本里, 这是因为LOAD DATA语句直到那时才具备“读取客户端数据文件以加载数据”的功能。(如果省略了关键字LOCAL, 就表示数据文件是保存在服务器主机上的, 而你必须拥有相应的服务器访问权限才能把有关文件里的数据加载到数据表里去。但是, 大多数MySQL用户都没有这种权限。)此外, MySQL 3.23.49及以后的版本虽然都具备有LOCAL机制, 但这一功能在默认情况下却可能处于禁用状态。如果LOAD DATA语句在执行时报告出错, 可以试试加上--local-infile选项再重新调用mysql程序, 比如:

```
% mysql --local-infile sampdb
mysql> LOAD DATA LOCAL INFILE 'member.txt' INTO TABLE member;
```

如果还不行, 就说明服务器端的LOCAL机制没有被激活。激活服务器端LOCAL机制的具体做法请参阅第11章内容。

mysqlimport是一个与LOAD DATA语句功能相同的命令行接口。当在操作系统的shell提示符下调用mysqlimport程序时, 它会生成一条LOAD DATA语句:

```
% mysqlimport --local sampdb member.txt
```

不过, 如果MySQL版本低于3.22.15, 就无法执行这个命令, 这是因为--local选项要求MySQL必须能够支持LOAD DATA LOCAL语句。此外, 与使用mysql客户程序时一样, 如果还需要设定连接参数, 请在命令行上把它们添加到数据库名称的前面。

就上面这条命令而言, mysqlimport程序将生成一条能够把member.txt文件里的数据值加载到member数据表里去的LOAD DATA语句。这是因为mysqlimport程序是根据数据文件的名字来确定与之对应的数据表的, 它将把文件名中第一个句点(.)之前的字符串用做数据表的名字。举例来说, 它会把member.txt和president.txt文件里的数据分别加载到member和president数据表里去。这就要求必须慎重选择数据文件的名字, 否则, mysqlimport程序会把数据错误地加载到别的数据表里去。例如, 你的想法是把member1.txt和member2.txt文件里的数据都加载到member数据表里去, 但mysqlimport却会认为你想把这两个文件分别加载到名为member1和member2的两个数据表里去。为了避免出现这种混乱, 可以把这两个文件命名为member.1.txt和member.2.txt, 或者是member.txt1和member.txt2。

在练习完上面介绍的这几种数据记录添加方法之后, 为了顺利进行后面的学习, 应该重新创建和加载sampdb数据库里的数据表, 把它们的内容恢复为原样。需要在操作系统的命令行上执行以下这些命令:

```
% mysql sampdb < create_president.sql
% mysql sampdb < insert_president.sql
% mysql sampdb < create_member.sql
% mysql sampdb < insert_member.sql
```

```
% mysql sampdb < create_student.sql
% mysql sampdb < insert_student.sql
% mysql sampdb < create_score.sql
% mysql sampdb < insert_score.sql
% mysql sampdb < create_event.sql
% mysql sampdb < insert_event.sql
% mysql sampdb < create_absence.sql
% mysql sampdb < insert_absence.sql
```

如果不喜欢输入这么多条命令，那么，在UNIX系统上，请执行下面这条命令：

```
% sh init_all_tables.sh sampdb
```

在Windows系统上，请执行下面这条命令：

```
C:\> init_all_tables.bat sampdb
```

如果还需要设定连接参数，请在命令行上把它们添加到数据库名称的前面。

#### 1.4.8 对信息进行检索

现在，数据表都已经创建出来并加载上数据了。下面，看看这些数据都能派上哪些用场。SELECT语句允许你以自己喜欢的方式对数据表里的信息进行检索和显示。比如说，可以像下面这样把整个数据表的内容都显示出来：

```
SELECT * FROM president;
```

也可以像下面这样只选取某一个数据行里的某一个数据列：

```
SELECT birth FROM president WHERE last_name = 'Eisenhower';
```

SELECT语句有几个子句（或者叫组成部分），它们的各种搭配能帮你查出最感兴趣的信息。这些子句可以很简单，也可以很复杂，由它们搭配出来的SELECT语句也会相应地变得简单或者复杂。不过，请大家放心，在这本书里，绝对没有长达数页而让大家必须花费1小时才能搞明白的查询命令。（如果看书时遇到长长的查询命令，我通常会跳过它们，我想你们也会如此。）

下面是SELECT语句的通用形式：

```
SELECT what to select
FROM table or tables
WHERE conditions that data must satisfy;
```

在写SELECT语句的时候，先指定想检索什么，再把必要的可选子句写出来。上面两个子句（FROM和WHERE）是最常见的，其他子句包括GROUP BY、ORDER BY和LIMIT等。需要指出的是，SQL语言对书写格式并没有严格的要求，所以在书写SELECT语句的时候，换行符的位置不必与本书示例中的一样。

FROM子句一般都少不了，但如果不需要给出数据表的名字，就不必把它写出来。比如说，下面这条语句只是计算表达式的值。因为这个计算不涉及任何数据表，所以也就没有必要把FROM子句写出来：

```
mysql> SELECT 2+2, 'Hello, world', VERSION();
+-----+-----+-----+
| 2+2 | Hello, world | VERSION() |
+-----+-----+-----+
| 4 | Hello, world | 4.0.4-beta-log |
+-----+-----+-----+
```

当的确需要使用FROM子句来指定将从哪个数据表检索数据的时候，还需要把想查看的数据列的名字列举出来。可以在SELECT语句里用一个星号(\*)来表示想查看所有的数据列。下面这条查询将把student数据表所有的数据列显示出来：

```
mysql> SELECT * FROM student;
+-----+-----+-----+
| name | sex | student_id |
+-----+-----+-----+
| Megan | F | 1 |
| Joseph | M | 2 |
| Kyle | M | 3 |
| Katie | F | 4 |
...
```

数据列将按它们在数据表里的存储先后顺序显示出来。这个顺序与用DESCRIBE student语句查看到的数据列排列顺序是一致的。(示例末尾处的省略号“...”表示这个查询所返回的数据行比这里列出的要多。)

也可以把自己想要查看的数据列的名字明确地列举出来。比如说，如果只想查看学生的姓名，就应该使用下面这条语句：

```
mysql> SELECT name FROM student;
+-----+
| name |
+-----+
| Megan |
| Joseph |
| Kyle |
| Katie |
...
```

如果需要列举多个数据列，请用逗号把它们分隔开。请看下面这条语句，它与SELECT \* FROM student是等价的，但它把各数据列的名字都明确地列举了出来：

```
mysql> SELECT name, sex, student_id FROM student;
+-----+-----+-----+
| name | sex | student_id |
+-----+-----+-----+
| Megan | F | 1 |
| Joseph | M | 2 |
| Kyle | M | 3 |
| Katie | F | 4 |
...
```



可以按任意顺序来列举数据列的名字:

```
SELECT name, student_id FROM student;
SELECT student_id, name FROM student;
```

只要愿意, 甚至还可以重复列举某些数据列的名字, 只是这样做通常没有多大的意义。

MySQL允许在一条SELECT语句里同时选取多个数据表里的数据列, 将在本小节后面对此进行讨论。

MySQL里的数据列名称不区分字母的大小写, 所以下面这些查询都是等价的:

```
SELECT name, student_id FROM student;
SELECT NAME, STUDENT_ID FROM student;
SELECT nAmE, sTuDeNt_Id FROM student;
```

但需要注意的是, 数据库和数据表的名字却可能区分字母的大小写——这取决于服务器主机上所使用的文件系统。比如说, Windows文件名不区分字母的大小写, 所以运行在Windows系统上的服务器也就不区分数据库和数据表名字的大小写; UNIX文件名区分字母的大小写, 所以运行在UNIX系统上的服务器就将区分数据库和数据表名字的大小写。(属于UNIX阵营的Mac OS X是个例外: HFS+文件系统不区分文件名的大小写, 但UFS文件系统却区分文件名的大小写。)如果想让MySQL服务器不区分数据库和数据表名字中的字母大小写, 请参阅第10.2.5节的内容。

### 1. 检索条件

要想让SELECT语句只把满足特定条件的记录检索出来, 就必须给它加上WHERE子句来设定数据行的检索条件。只有这样, 才能有选择地把数据列的取值满足特定要求的那些数据行挑选出来。可以针对任何类型的值进行查找。比如说, 可以针对数值进行搜索:

```
mysql> SELECT * FROM score WHERE score > 95;
```

student_id	event_id	score
5	3	97
18	3	96
1	6	100
5	6	97
11	6	98
16	6	98

也可以针对字符串值进行查找(注意: 字符串的比较操作通常不区分字母的大小写):

```
mysql> SELECT last_name, first_name FROM president
-> WHERE last_name='ROOSEVELT';
```

last_name	first_name
Roosevelt	Theodore
Roosevelt	Franklin D.

```
mysql> SELECT last_name, first_name FROM president
-> WHERE last_name='roosevelt';
```

```
+-----+-----+
| last_name | first_name |
+-----+-----+
| Roosevelt | Theodore   |
| Roosevelt | Franklin D. |
+-----+-----+
```

还可以针对日期值进行查找:

```
mysql> SELECT last_name, first_name, birth FROM president
-> WHERE birth < '1750-1-1';
```

```
+-----+-----+-----+
| last_name | first_name | birth      |
+-----+-----+-----+
| Washington | George     | 1732-02-22 |
| Adams      | John       | 1735-10-30 |
| Jefferson  | Thomas     | 1743-04-13 |
+-----+-----+-----+
```

甚至还能针对不同类型的值的组合情况进行查找:

```
mysql> SELECT last_name, first_name, birth, state FROM president
-> WHERE birth < '1750-1-1' AND (state='VA' OR state='MA');
```

```
+-----+-----+-----+-----+
| last_name | first_name | birth      | state |
+-----+-----+-----+-----+
| Washington | George     | 1732-02-22 | VA    |
| Adams      | John       | 1735-10-30 | MA    |
| Jefferson  | Thomas     | 1743-04-13 | VA    |
+-----+-----+-----+-----+
```

WHERE子句里的表达式允许使用算术操作符(见表1-1)、比较操作符(见表1-2)和逻辑操作符(见表1-3),还允许使用括号。这些表达式允许使用常数、数据表的数据列以及函数调用进行运算。虽然在该教程的查询示例里用到的MySQL函数不算很多,但它们的数量其实并不少。MySQL函数的完整清单请参阅附录C。

表1-1 算术操作符

操 作 符	含 义
+	加法
-	减法
*	乘法
/	除法
%	求余(整数除法后的剩余部分)

表1-2 比较操作符

操 作 符	含 义
<	小于
<=	小于或等于（不大于）
=	等于
<=>	等于（能够对NULL值进行比较）
!= 或 <>	不等于
>=	大于或等于（不小于）
>	大于

表1-3 逻辑操作符

操 作 符	含 义
AND	逻辑与
OR	逻辑或
XOR	逻辑异或
NOT	逻辑非

如果需要在查询语句里使用逻辑操作符，千万要注意一点：逻辑AND操作符与人们日常生活中所说的“和”在含义上是不一样的。举个例子，假设有把“出生于弗吉尼亚州和出生于马萨诸塞州的总统”给查出来。这个问题里有一个“和”字，头脑简单的人会不假思索地写出一条下面这样的查询命令：

```
mysql> SELECT last_name, first_name, state FROM president
-> WHERE state='VA' AND state='MA';
Empty set (0.36 sec)
```

这个查询的结果集是空的，没有把我们想要的东西找出来。为什么会这样呢？因为这条查询的真正含义是“把同时出生于弗吉尼亚州和出生于马萨诸塞州的总统”给找出来，而这种情况是不可能出现的。在日常生活里，可以用“和”来表达查询条件，但在SQL里，却必须把这两个条件用逻辑或操作符OR连接在一起，即：

```
mysql> SELECT last_name, first_name, state FROM president
-> WHERE state='VA' OR state='MA';
```

```
+-----+-----+-----+
| last_name | first_name | state |
+-----+-----+-----+
| Washington | George    | VA    |
| Adams      | John      | MA    |
| Jefferson  | Thomas    | VA    |
| Madison    | James     | VA    |
| Monroe     | James     | VA    |
| Adams      | John Quincy | MA    |
| Harrison   | William H. | VA    |
| Tyler      | John      | VA    |
+-----+-----+-----+
```

Taylor	Zachary	VA	
Wilson	Woodrow	VA	
Kennedy	John F	MA	
Bush	George H.W.	MA	
+-----+-----+-----+			

请大家务必注意日常语言与SQL之间的这类差异——在为自己编写查询命令时要注意，在为其他人编写查询命令时更要注意。一定要仔细听取别人对查询情况的描述，然后根据对方的描述正确地选用适当的SQL逻辑操作符。以上面的查询为例，它正确的自然语言表述形式应该是：“把出生于弗吉尼亚州或者出生于马萨诸塞州的总统给找出来。”

## 2. NULL值

NULL是一个很特别的值。它的含义是“无数据”，所以不能用它与“有数据”的值进行运算或者比较。如果试图用普通的算术操作符和比较操作符对NULL值进行操作，其结果将是不可预料的：

```
mysql> SELECT NULL < 0, NULL = 0, NULL != 0, NULL > 0;
+-----+-----+-----+-----+
| NULL < 0 | NULL = 0 | NULL != 0 | NULL > 0 |
+-----+-----+-----+-----+
| NULL | NULL | NULL | NULL |
+-----+-----+-----+-----+
```

事实上，甚至不应该把NULL值与它本身进行比较，因为两个表示“无数据”的未知值的比较结果也将是不可预料的：

```
mysql> SELECT NULL = NULL, NULL != NULL;
+-----+-----+
| NULL = NULL | NULL != NULL |
+-----+-----+
| NULL | NULL |
+-----+-----+
```

如果需要对NULL值进行查找，就必须使用一种特殊的语法。不能使用“=”或者“!=”来测试它们是相等还是不相等，必须使用“IS NULL”或“IS NOT NULL”来进行判断。比如说，如果想把目前仍然健在的美国总统给查出来，就应该使用一条下面这样的查询命令——因为这些总统的逝世日期在president数据表里是用NULL值来表示的：

```
mysql> SELECT last_name, first_name FROM president WHERE death IS NULL;
+-----+-----+
| last_name | first_name |
+-----+-----+
| Ford      | Gerald R   |
| Carter    | James E.   |
| Reagan    | Ronald W.   |
| Bush      | George H.W. |
| Clinton   | William J.  |
| Bush      | George W.   |
+-----+-----+
```



同样，如果想把没有姓名后缀的美国总统给查出来，就应该在检索条件里使用“IS NOT NULL”进行判断：

```
mysql> SELECT last_name, first_name, suffix
      -> FROM president WHERE suffix IS NOT NULL;
+-----+-----+-----+
| last_name | first_name | suffix |
+-----+-----+-----+
| Carter   | James E.   | Jr.    |
+-----+-----+-----+
```

MySQL 3.23及以后的版本里多了一个专用的MySQL比较操作符“<=>”，它能完成NULL值与NULL值之间的比较。可以用这个操作符把上面的两条查询命令分别改写为：

```
mysql> SELECT last_name, first_name FROM president WHERE death <=> NULL;
+-----+-----+
| last_name | first_name |
+-----+-----+
| Ford      | Gerald R   |
| Carter    | James E.   |
| Reagan    | Ronald W.   |
| Bush      | George H.W. |
| Clinton   | William J. |
| Bush      | George W.   |
+-----+-----+

mysql> SELECT last_name, first_name, suffix
      -> FROM president WHERE NOT (suffix <=> NULL);
+-----+-----+-----+
| last_name | first_name | suffix |
+-----+-----+-----+
| Carter    | James E.   | Jr.    |
+-----+-----+-----+
```

### 3. 对查询结果进行排序

MySQL用户迟早会注意到这样一种情况：如果创建了一个数据表并往里面加载了一些数据记录，当发出一条SELECT \* FROM *tbl\_name*语句时，数据记录在查询结果中的先后顺序通常与它们当初被插入时的先后顺序一致。这很符合人们的思维习惯，因此，人们很自然地把“数据记录在查询结果中的先后顺序与它们当初被插入时的先后顺序相同”这种假设接受为一个可以信赖的原则。但这种假设是不正确的。例如，如果在加载完数据表的初始数据之后又删除并插入了一些数据行，这些操作往往会改变有关数据行在服务器所返回的数据表检索结果中的先后顺序。（数据记录删除操作会在数据表里留下一些“空洞”，而MySQL会用以后插入的新记录来尽量填补这些“空洞”。）

可以真正信赖的原则是：从服务器返回的数据行的先后顺序没有任何保证，除非事先进行了设定。如果想让查询结果按所希望的先后顺序来显示，就必须给查询命令增加一条ORDER

BY子句以设定其查询结果的排列顺序。下面这条查询命令将把美国总统们的姓名按他们姓氏的字母顺序排列并显示：

```
mysql> SELECT last_name, first_name FROM president
      -> ORDER BY last_name;
+-----+-----+
| last_name | first_name |
+-----+-----+
| Adams    | John      |
| Adams    | John Quincy |
| Arthur   | Chester A. |
| Buchanan | James     |
...
```

在出现于ORDER BY子句中的数据列名字的后面加上关键字ASC或DESC，就能使查询结果中的数据记录按指定数据列的升序或者降序排列，比如说，如果想让美国总统们的姓名按他们姓氏的逆序（降序）排列显示，就应该像下面这样加上DESC关键字：

```
mysql> SELECT last_name, first_name FROM president
      -> ORDER BY last_name DESC;
+-----+-----+
| last_name | first_name |
+-----+-----+
| Wilson    | Woodrow    |
| Washington | George     |
| Van Buren | Martin     |
| Tyler     | John       |
...
```

如果没有在出现于ORDER BY子句中的数据列名字的后面加上关键字ASC或DESC，MySQL就将默认地把查询结果中的数据记录按升序进行排列和显示。

MySQL允许对查询结果按多个数据列进行排序，而每一个数据列又都可以互不影响地分别按升序或降序进行排列。下面这条针对president数据表的查询命令将把查询结果中的数据行按总统出生地所在州的逆序进行排列，而出生地所在州相同的总统姓名又将按其姓氏的升序排列：

```
mysql> SELECT last_name, first_name, state FROM president
      -> ORDER BY state DESC, last_name ASC;
+-----+-----+-----+
| last_name | first_name | state |
+-----+-----+-----+
| Arthur   | Chester A. | VT    |
| Coolidge | Calvin     | VT    |
| Harrison | William H. | VA    |
| Jefferson | Thomas     | VA    |
| Madison  | James     | VA    |
| Monroe   | James     | VA    |
| Taylor   | Zachary    | VA    |
```

Tyler	John	VA
Washington	George	VA
Wilson	Woodrow	VA
Eisenhower	Dwight D.	TX
Johnson	Lyndon B.	TX

...

如果进行排序的数据列里可能包含有NULL值，那些包含有NULL值的数据行将出现在查询结果中的什么位置呢？这要视具体情况而定。从MySQL 4.0.2开始，包含有NULL值的数据行总是出现在查询结果的开头——哪怕把排序顺序设定为DESC也将如此。而在此之前，如果设定的排序顺序是ASC，它们将出现在查询结果的开头；如果设定的排序顺序是DESC，它们将出现在查询结果的末尾。

如果想让包含有NULL值的数据行出现在查询结果的末尾，就必须额外增加一个排序数据列以区分NULL值和非NULL值。比如说，如果想按逝世日期对总统们的姓名进行查询和排序，并想让仍然健在（即逝世日期等于NULL）的总统们的姓名出现在查询结果的开头，那就应该使用一条下面这样的查询命令：

```
mysql> SELECT last_name, first_name, death FROM president
        -> ORDER BY IF(death IS NULL,0,1), death;
```

last_name	first_name	death
Ford	Gerald R	NULL
Carter	James E.	NULL
Reagan	Ronald W.	NULL
Bush	George H.W.	NULL
Clinton	William J.	NULL
Bush	George W.	NULL
Washington	George	1799-12-14
Adams	John	1826-07-04
...		
Truman	Harry S.	1972-12-26
Johnson	Lyndon B.	1973-01-22
Nixon	Richard M	1994-04-22

如果想让仍然健在的总统们的姓名出现在查询结果的末尾，那就应该使用一条下面这样的查询命令：

```
mysql> SELECT last_name, first_name, death FROM president
        -> ORDER BY IF(death IS NULL,1,0), death;
```

last_name	first_name	death
Washington	George	1799-12-14
Adams	John	1826-07-04
...		

Truman	Harry S.	1972-12-26	
Johnson	Lyndon B.	1973-01-22	
Nixon	Richard M	1994-04-22	
Ford	Gerald R	NULL	
Carter	James E.	NULL	
Reagan	Ronald W.	NULL	
Bush	George H.W.	NULL	
Clinton	William J.	NULL	
Bush	George W.	NULL	
+-----+-----+-----+			

IF()函数的作用是对紧随其后的表达式进行求值，再根据表达式求值结果的真假返回它的第2个参数或第3个参数的值。在刚才的第一个查询里，在遇到NULL值的时候，IF()函数的求值结果将是0；在遇到非NULL值的时候，IF()函数的求值结果将是1；最终效果是把包含有NULL值的数据行全都放在了非NULL值的数据行的前面。在第二个查询里，正好相反，于是最终效果就变成了把包含有NULL值的数据行全都放在了非NULL值的数据行的后面。这个策略适用于MySQL 3.23.2及以后的版本，因为3.23.2版本是允许ORDER BY子句对表达式进行求值的第一个版本。

#### 4. 限制查询结果中的数据行个数

查询结果往往由很多个数据行构成，如果只想看到其中的一小部分，可以给查询命令增加一条LIMIT子句来限制查询结果中的数据行个数——它与ORDER BY子句联合使用的效果往往更佳。MySQL允许给查询结果中的数据行个数设置一个上限（比如说 $n$ 个数据行），如果查询结果里的数据行超过了这个数字，就只显示前 $n$ 个数据行。下面这个查询将把出生日期最早的前5个总统列举出来：

```
mysql> SELECT last_name, first_name, birth FROM president
-> ORDER BY birth LIMIT 5;
```

last_name	first_name	birth	
Washington	George	1732-02-22	
Adams	John	1735-10-30	
Jefferson	Thomas	1743-04-13	
Madison	James	1751-03-16	
Monroe	James	1758-04-28	

如果使用了ORDER BY birth DESC，即按逆序来排列查询结果，则找出来的就将是出生日期最晚的前5个总统：

```
mysql> SELECT last_name, first_name, birth FROM president
-> ORDER BY birth DESC LIMIT 5;
```

last_name	first_name	birth	
-----------	------------	-------	--



Clinton	William J.	1946-08-19
Bush	George W.	1946-07-06
Carter	James E.	1924-10-01
Bush	George H.W.	1924-06-12
Kennedy	John F.	1917-05-29

LIMIT还允许从查询结果的中间部分抽出一部分数据记录。此时，需要设定两个值，第一个值给出了要在查询结果的开头部分跳过去的的数据记录个数，第二个值则是需要返回的数据记录的个数。下面的查询与前一个很相似，但它返回的是跳过前10个数据记录之后的5个数据记录：

```
mysql> SELECT last_name, first_name, birth FROM president
       -> ORDER BY birth LIMIT 10, 5;
```

last_name	first_name	birth
Tyler	John	1790-03-29
Buchanan	James	1791-04-23
Polk	James K.	1795-11-02
Fillmore	Millard	1800-01-07
Pierce	Franklin	1804-11-23

如果想从president数据表里随机抽出一条数据记录，可以联合使用LIMIT和ORDER BY RAND()子句，如下所示：

```
mysql> SELECT last_name, first_name FROM president
       -> ORDER BY RAND() LIMIT 1;
```

last_name	first_name
McKinley	William

这种根据某个公式的计算结果来进行排序的做法适用于MySQL 3.23.2及以后的版本。在此之前，必须采用额外生成一个内容为随机数字的数据列的办法才能解决这个问题，关于这个问题的详细讨论请参阅第4.2.2节的内容。

### 5. 对输出列进行求值和命名

前面给出的查询示例的输出结果大都是直接检索自数据表的数据值。MySQL还允许把表达式的计算结果当做输出列的值。表达式可以很简单，也可以很复杂。例如，下面这个查询有两个输出列，前一个输出列对应着一个非常简单的表达式（一个常数），而后一个输出列则对应着一个使用了多个算术操作符和两个函数调用的复杂表达式：

```
mysql> SELECT 17, FORMAT(SQRT(3*3+4*4),0);
```

17	FORMAT(SQRT(3*3+4*4),0)
17	5

数据表里的数据列名字也可以用在表达式里，如下所示：

```
mysql> SELECT CONCAT(first_name, ' ', last_name), CONCAT(city, ', ', state)
-> FROM president;

+-----+-----+
| CONCAT(first_name, ' ', last_name) | CONCAT(city, ', ', state) |
+-----+-----+
| George Washington                  | Wakefield, VA             |
| John Adams                        | Braintree, MA             |
| Thomas Jefferson                  | Albemarle County, VA      |
| James Madison                     | Port Conway, VA           |
...
```

在这个查询里，对输出列的格式进行了设置：总统们的名字和姓氏合二为一，成为了一个以空格分隔的字符串，他们的出生城市和出生州则被合并为一个以逗号(,)分隔的字符串。

如果输出列的值是某个表达式的计算结果，这个表达式就会成为这个输出列的名字并被用做它在输出结果中的标题。如果表达式很长（例如上面这个查询示例），就会使输出列的宽度变得很大。为解决这一问题，可以利用AS name结构给输出列另外取一个名字，称之为（输出）列的别名（alias）。比如说，可以像下面这样把上面查询的输出结果改写得更有意义：

```
mysql> SELECT CONCAT(first_name, ' ', last_name) AS Name,
-> CONCAT(city, ', ', state) AS Birthplace
-> FROM president;

+-----+-----+
| Name                                | Birthplace                  |
+-----+-----+
| George Washington                  | Wakefield, VA              |
| John Adams                        | Braintree, MA              |
| Thomas Jefferson                  | Albemarle County, VA      |
| James Madison                     | Port Conway, VA           |
...
```

如果输出列的别名里包含有空格，就必须把它放到一对引号里，如下所示：

```
mysql> SELECT CONCAT(first_name, ' ', last_name) AS 'President Name',
-> CONCAT(city, ', ', state) AS 'Place of Birth'
-> FROM president;

+-----+-----+
| President Name                      | Place of Birth              |
+-----+-----+
| George Washington                  | Wakefield, VA              |
| John Adams                        | Braintree, MA              |
| Thomas Jefferson                  | Albemarle County, VA      |
| James Madison                     | Port Conway, VA           |
...
```

## 6. 与日期有关的问题

在与MySQL里的日期打交道的时候，千万要记住年份总是出现在最前面（英语习惯把年份

放在日期的最后，如“July 27, 2002”。——译者注）。2002年7月27日将被表示为 '2002-07-27'，而不是像日常生活中那样被表示为 '07-27-2002' 或 '27-07-2002'。

MySQL提供了几种对日期进行操作的方法，比较常见的有：

- 按日期进行排序。（已经介绍过几个这方面的例子了。）
- 查找某个日期或者某个日期范围。
- 提取日期值中的年、月、日等组成部分。
- 计算两个日期之间的时间距离。
- 用一个日期加上或减去一个时间间隔以求出另一个日期。

下面是一些与日期有关的查询示例。

如果查询某特定日期，可以利用具体的日期值，也可以与另一个日期值进行比较，即把一个DATE类型的数据列与感兴趣的那个日期值进行比较，如下所示：

```
mysql> SELECT * FROM event WHERE date = '2002-10-01';
+-----+-----+-----+
| date      | type | event_id |
+-----+-----+-----+
| 2002-10-01 | T    | 6        |
+-----+-----+-----+

mysql> SELECT last_name, first_name, death
-> FROM president
-> WHERE death >= '1970-01-01' AND death < '1980-01-01';
+-----+-----+-----+
| last_name | first_name | death      |
+-----+-----+-----+
| Truman   | Harry S.   | 1972-12-26 |
| Johnson  | Lyndon B.  | 1973-01-22 |
+-----+-----+-----+
```

日期中的年、月、日三部分可以用函数YEAR()、MONTH()、DAYOFMONTH()分别分离出来。比如说，下面这个查询将把3月出生的美国总统查出来：

```
mysql> SELECT last_name, first_name, birth
-> FROM president WHERE MONTH(birth) = 3;
+-----+-----+-----+
| last_name | first_name | birth      |
+-----+-----+-----+
| Madison   | James     | 1751-03-16 |
| Jackson   | Andrew    | 1767-03-15 |
| Tyler     | John      | 1790-03-29 |
| Cleveland | Grover     | 1837-03-18 |
+-----+-----+-----+
```

在这个查询里，还可以直接使用3月的英文“March”：



```
mysql> SELECT last_name, first_name, birth
        -> FROM president WHERE MONTHNAME(birth) = 'March';
```

last_name	first_name	birth
Madison	James	1751-03-16
Jackson	Andrew	1767-03-15
Tyler	John	1790-03-29
Cleveland	Grover	1837-03-18

再进一步，把MONTH()和DAYOFMONTH()函数结合起来使用，就能把3月29日出生的总统给查出来：

```
mysql> SELECT last_name, first_name, birth
        -> FROM president WHERE MONTH(birth) = 3 AND DAYOFMONTH(birth) = 29;
```

last_name	first_name	birth
Tyler	John	1790-03-29

很多报纸的娱乐版都有“今日出生的名人”之类的栏目，而上面的查询就能生成一份这样的名单。不过，如果查询与“今天”有关，那大可不必像前面的例子那样使用一个具体的日期值——MySQL提供了一个CURDATE()函数，它的返回值永远是“今天”的日期值。于是，“今日出生的总统”完全可以用下面这个查询找出来：

```
SELECT last_name, first_name, birth
FROM president WHERE MONTH(birth) = MONTH(CURDATE())
AND DAYOFMONTH(birth) = DAYOFMONTH(CURDATE());
```

如果想知道两个日期值之间的时间间隔，进行减法运算即可。比如说，如果想知道哪位总统的寿命最长，就得用他们的逝世日期减去他们的出生日期。具体做法是：先用TO\_DAYS()函数把总统们的出生日期和逝世日期转换为天数，再求出二者的差。如下所示：

```
mysql> SELECT last_name, first_name, birth, death,
        -> TO_DAYS(death) - TO_DAYS(birth) AS age
        -> FROM president WHERE death IS NOT NULL
        -> ORDER BY age DESC LIMIT 5;
```

last_name	first_name	birth	death	age
Adams	John	1735-10-30	1826-07-04	33119
Hoover	Herbert C.	1874-08-10	1964-10-20	32943
Truman	Harry S.	1884-05-08	1972-12-26	32373
Madison	James	1751-03-16	1836-06-28	31150
Jefferson	Thomas	1743-04-13	1826-07-04	30397



用天数来表示的age（岁数）不符合日常习惯，可以把它们转换为年数。用天数除以365（在提到岁数的时候，人们习惯用一个整数而不是小数，所以使用FLOOR()函数来消去除法运算结果中的小数部分以得到一个整数）：

```
mysql> SELECT last_name, first_name, birth, death,
-> FLOOR((TO_DAYS(death) - TO_DAYS(birth))/365) AS age
-> FROM president WHERE death IS NOT NULL
-> ORDER BY age DESC LIMIT 5;
```

last_name	first_name	birth	death	age
Adams	John	1735-10-30	1826-07-04	90
Hoover	Herbert C.	1874-08-10	1964-10-20	90
Truman	Harry S.	1884-05-08	1972-12-26	88
Madison	James	1751-03-16	1836-06-28	85
Jefferson	Thomas	1743-04-13	1826-07-04	83

就这个例子而言，求出来的age值恰好与总统们逝世时的年岁相同。但千万不要因此而认为日期值的这种转换和计算永远都会产生正确的结果——一年并不是都有365天，还有闰年。按日常习惯，应该这样来计算总统们的寿命：先把逝世日期和出生日期中的年份相减；然后，如果逝世日期的日历值早于出生日期的日历值，就再减去1。如下所示：

```
mysql> SELECT last_name, first_name, birth, death,
-> (YEAR(death) - YEAR(birth)) - IF(RIGHT(death,5) < RIGHT(birth,5),1,0)
-> AS age
-> FROM president WHERE death IS NOT NULL
-> ORDER BY age DESC LIMIT 5;
```

last_name	first_name	birth	death	age
Adams	John	1735-10-30	1826-07-04	90
Hoover	Herbert C.	1874-08-10	1964-10-20	90
Truman	Harry S.	1884-05-08	1972-12-26	88
Madison	James	1751-03-16	1836-06-28	85
Jefferson	Thomas	1743-04-13	1826-07-04	83

日历值的比较是通过IF()表达式对日期值后5个字符的减法结果进行判断而实现的。这样做有两个理由：其一，如果把日期值传到一个字符串函数（如本例的RIGHT()，返回字符串最右端的n个字符）里，MySQL就将把它们看做是字符串；其二，在MySQL里，日期的组成部分（年、月、日）都有固定的位数，小于10的月份或日期的前面将补上一个“0”——如果不是这样，上面语句中的比较操作就无法正确进行。

日期值的减法运算还能帮我们计算出距某特定日期还有多长的时间，而这正是我们用来找出需要在近期交纳会费的“美国历史研究会”会员的办法：用某会员资格的失效日期减去“今

天”的日期，若其结果小于某个阈值，就表明这位会员需要续费了。下面这个查询将把需要在60天以内续交会费的会员给查出来：

```
SELECT last_name, first_name, expiration FROM member
WHERE (TO_DAYS(expiration) - TO_DAYS(CURDATE())) < 60;
```

日期值的加减法必须用DATE\_ADD()和DATE\_SUB()函数来完成。这两个函数的输入参数是一个日期值和一个时间间隔值，返回结果则是一个新日期值。请看下面的例子：

```
mysql> SELECT DATE_ADD('1970-1-1', INTERVAL 10 YEAR);
+-----+
| DATE_ADD('1970-1-1', INTERVAL 10 YEAR) |
+-----+
| 1980-01-01                               |
+-----+
mysql> SELECT DATE_SUB('1970-1-1', INTERVAL 10 YEAR);
+-----+
| DATE_SUB('1970-1-1', INTERVAL 10 YEAR) |
+-----+
| 1960-01-01                               |
+-----+
```

前面有一个用来查询“哪些美国总统逝世于20世纪70年代”的示例，其中用了两个确切的日期值来表示时间的起止点。利用日期值的加减法运算，原来的查询可以改写为：起点日期仍使用一个确切的日期，但终点日期却是通过起点日期加上一个时间间隔而计算出来的。如下所示：

```
mysql> SELECT last_name, first_name, death
-> FROM president
-> WHERE death >= '1970-1-1'
-> AND death < DATE_ADD('1970-1-1', INTERVAL 10 YEAR);
+-----+-----+-----+
| last_name | first_name | death      |
+-----+-----+-----+
| Truman   | Harry S.   | 1972-12-26 |
| Johnson  | Lyndon B.  | 1973-01-22 |
+-----+-----+-----+
```

用来查找“需要在近期交纳会费的会员”的查询可以改写为：

```
SELECT last_name, first_name, expiration FROM member
WHERE expiration < DATE_ADD(CURDATE(), INTERVAL 60 DAY);
```

在本章前面的内容里，曾给出一个牙医诊所用来查找“哪些患者没来复查”的查询：

```
SELECT last_name, first_name, last_visit FROM patient
WHERE last_visit < DATE_SUB(CURDATE(), INTERVAL 6 MONTH);
```

你当时也许还看不懂这个查询，现在总该明白了吧？

## 7. 模式匹配

MySQL支持模式匹配操作，这使我们能够在没有给出精确比较值的情况下把有关的数据记录查出来。模式匹配需要使用特殊的操作符（LIKE 和 NOT LIKE），还需要提供一个包含通配符的字符串。下划线字符“\_”只能匹配单个的任意字符，百分号字符“%”则能匹配任何一个字符序列（包括空序列在内）。用LIKE和NOT LIKE操作符进行的模式匹配不区分字母的大小写情况。

下面这个查询将把姓氏以字母“W”或“w”开头的总统姓名查找出来：

```
mysql> SELECT last_name, first_name FROM president
-> WHERE last_name LIKE 'W%';
```

last_name	first_name
Washington	George
Wilson	Woodrow

请大家再来看一个查询，它在使用模式匹配功能时出现了失误：

```
mysql> SELECT last_name, first_name FROM president
-> WHERE last_name = 'W%';
Empty set (0.00 sec)
```

打算进行模式匹配却错误地使用了一个算术比较操作符是一种很容易发生的失误。在这种错误情况里，WHERE子句里的条件表达式本身并没有出错，但整个查询的含义却变成了“把姓氏是‘W%’或‘w%’的总统找出来”。

下面这个查询将把姓氏里有“W”或“w”字母（并不仅限于姓氏的第一个字母）的总统姓名列出来：

```
mysql> SELECT last_name, first_name FROM president
-> WHERE last_name LIKE '%W%';
```

last_name	first_name
Washington	George
Wilson	Woodrow
Eisenhower	Dwight D.

下面这个查询将把姓氏由且仅由四个字母构成的总统姓名给查出来：

```
mysql> SELECT last_name, first_name FROM president
-> WHERE last_name LIKE '____';
```

last_name	first_name
Polk	James K.

Taft	William H.	
Ford	Gerald R	
Bush	George H.W.	
Bush	George W.	

+-----+-----+

MySQL还提供了基于正则表达式 (regular expression) 的模式匹配功能, 这些内容将在附录C对REGEXP操作符进行讨论时加以介绍。

#### 8. 设置和使用SQL变量

MySQL 3.23.6及以后的版本允许使用查询结果来设置变量, 这使我们能够方便地把一些值保存起来以供今后使用。比如说, 如果想知道有哪些总统出生在Andrew Jackson总统之前, 可以这样做: 先检索出他的出生日期并保存到一个变量里, 再把出生日期早于这个变量值的那些总统查找出来<sup>①</sup>:

```
mysql> SELECT @birth := birth FROM president
      -> WHERE last_name = 'Jackson' AND first_name = 'Andrew';
+-----+
| @birth := birth |
+-----+
| 1767-03-15      |
+-----+

mysql> SELECT last_name, first_name, birth FROM president
      -> WHERE birth < @birth ORDER BY birth;
+-----+-----+-----+
| last_name | first_name | birth      |
+-----+-----+-----+
| Washington | George    | 1732-02-22 |
| Adams      | John      | 1735-10-30 |
| Jefferson  | Thomas    | 1743-04-13 |
| Madison    | James     | 1751-03-16 |
| Monroe     | James     | 1758-04-28 |
+-----+-----+-----+
```

变量的命名语法是@name, 赋值语法是在SELECT语句里使用一个“@name := value”形式的表达式。因此, 上面的第一个查询负责把Andrew Jackson总统的出生日期查出来并把它赋值给一个名为“@birth”的变量 (这条SELECT语句的查询结果仍会显示, 把查询结果赋值给一个变量并不会使该查询的输出结果不显示); 第二个查询负责把出生日期早于@birth变量值的总统们的记录给查出来。

SET语句也能用来对变量进行赋值, 此时, =和:=都可以用做赋值操作符。如下所示:

<sup>①</sup> 这个问题完全可以用一个使用了结合 (join, 也叫“关联”) 的查询命令来解决, 但我们现在还没有讲到如何编写结合的方法。而且, 有些问题用变量来解决往往更容易。



```
mysql> SET @one_week_ago = DATE_SUB(CURDATE(),INTERVAL 7 DAY);
mysql> SELECT CURDATE(), @one_week_ago;
+-----+-----+
| CURDATE() | @one_week_ago |
+-----+-----+
| 2002-09-03 | 2002-08-27    |
+-----+-----+
```

### 9. 生成统计信息

MySQL最有用的功能之一是它能够依据大量未经加工的数据生成多种统计汇总信息。大家知道，单纯依靠人工手段来生成统计信息是一项既艰苦又耗时还容易出错的工作。如果大家能掌握使用MySQL来生成各种统计信息的技巧，它就会成为最具威力的信息处理工具。

找出一组数据里到底有多少种不同的取值是一项比较常见的统计工作，而关键字DISTINCT可以把查询结果中重复出现的数据行清除掉。比如说，下面的查询将把美国历届总统的出生地所在州不加重复地列举出来：

```
mysql> SELECT DISTINCT state FROM president ORDER BY state;
+-----+
| state |
+-----+
| AR    |
| CA    |
| CT    |
| GA    |
| IA    |
| IL    |
| KY    |
| MA    |
| MO    |
| NC    |
| NE    |
| NH    |
| NJ    |
| NY    |
| OH    |
| PA    |
| SC    |
| TX    |
| VA    |
| VT    |
+-----+
```

另一项比较常见的统计工作是利用COUNT()函数来统计有关记录的个数。COUNT(\*)可以统计查询到底选取了多少个数据行。如果查询语句没有WHERE子句，COUNT(\*)就会把数据表里总共有多少个数据行的情况统计出来。下面这个查询可以统计“美国历史研究会”现在共有多少

少名会员：

```
mysql> SELECT COUNT(*) FROM member;
```

```
+-----+
| COUNT(*) |
+-----+
|      102 |
+-----+
```

如果查询语句带有WHERE子句，COUNT(\*)就将统计出该子句到底匹配了多少个数据行。比如说，下面这个查询可以统计出到目前为止已经进行过多少次考试：

```
mysql> SELECT COUNT(*) FROM event WHERE type = 'Q';
```

```
+-----+
| COUNT(*) |
+-----+
|         4 |
+-----+
```

COUNT(\*)的统计结果是所选取数据行的总数，而COUNT(col\_name)值则只统计非NULL值的个数。二者之间的区别很容易从下面这个查询看出来：

```
mysql> SELECT COUNT(*),COUNT(email),COUNT(expiration) FROM member;
```

```
+-----+-----+-----+
| COUNT(*) | COUNT(email) | COUNT(expiration) |
+-----+-----+-----+
|      102 |           80 |           96 |
+-----+-----+-----+
```

从上面的查询结果可以知道，member数据表目前共有102条记录，其中只有80条在email数据列里有值。我们还可以推断出研究会目前有6名终身会员——expiration数据列里的NULL值表示这是一名终身会员，因为102条记录里有96条记录的expiration数据列里有非NULL值，所以剩下的6条记录就必然属于那些终身会员。

MySQL 3.23.2及以后的版本允许把COUNT()和DISTINCT联合起来使用，这使我们能够轻松地统计出查询结果里到底有多少种不同的值。比如说，如果想知道美国总共有多少个州曾经出生过总统，可以使用下面这个查询：

```
mysql> SELECT COUNT(DISTINCT state) FROM president;
```

```
+-----+
| COUNT(DISTINCT state) |
+-----+
|                20 |
+-----+
```

可以对某个数据列进行全面的统计，也可以对该数据列做分门别类的统计。比如说，如果想知道班级里总共有多少名学生，可以使用下面这个查询：

```
mysql> SELECT COUNT(*) FROM student;
```

```
+-----+
| COUNT(*) |
+-----+
|      31  |
+-----+
```

但如果想知道班级里分别有多少名男生和女生又该怎么办呢？一种办法是分别对两种性别进行统计，如下所示：

```
mysql> SELECT COUNT(*) FROM student WHERE sex='f';
```

```
+-----+
| COUNT(*) |
+-----+
|      15  |
+-----+
```

```
mysql> SELECT COUNT(*) FROM student WHERE sex='m';
```

```
+-----+
| COUNT(*) |
+-----+
|      16  |
+-----+
```

这个办法管用，但比较麻烦，如果数据列的不同取值有很多的话，这个办法就算管用也不适用了。以统计出生于美国不同州里的总统人数为例：必须先把有多少个不同的州出生过总统的情况统计出来（`SELECT DISTINCT state FROM president`），然后再用一系列“`SELECT COUNT(*)`”语句去分别统计出生于各州的总统人数。如此麻烦的事情是不会有几个人情愿去做的。

值得庆幸的是，MySQL只用一个查询就能把某数据列里的不同取值分别出现过多少次的情况给统计出来。还是用分别统计男、女生人数的例子，可以把原来的两个查询改写为下面这样的一个查询：

```
mysql> SELECT sex, COUNT(*) FROM student GROUP BY sex;
```

```
+-----+-----+
| sex | COUNT(*) |
+-----+-----+
| F   |      15  |
| M   |      16  |
+-----+-----+
```

分别统计出生于各州的美国总统人数的问题也可以用类似的办法来解决，如下所示：

```
mysql> SELECT state, COUNT(*) FROM president GROUP BY state;
```

```
+-----+-----+
| state | COUNT(*) |
+-----+-----+
| AR    |         1 |
| CA    |         1 |
+-----+-----+
```

CT		1	
GA		1	
IA		1	
IL		1	
KY		1	
MA		4	
MO		1	
NC		2	
NE		1	
NH		1	
NJ		1	
NY		4	
OH		7	
PA		1	
SC		1	
TX		2	
VA		8	
VT		2	
+-----+-----+			

如果需要进行这种分门别类的统计，GROUP BY子句就必不可少，它的作用是让MySQL知道在进行统计之前应该先如何对有关的数据记录进行分类。如果忘了加上这个子句，就会收到一条出错信息。

与反复使用多个彼此近似的查询来分别统计某数据列不同取值出现次数的做法相比，把COUNT(\*)函数与GROUP BY子句相结合的办法有很多优点，主要表现在：

- 在开始统计之前，不必知道将被统计的数据列里到底有多少种不同的取值。
- 只需使用一个而不是几个查询。
- 因为只用一个查询就能把所有的结果都查出来，所以还能对输出进行排序。

前两项优点的重要性体现在它们有助于简化查询语句的书写工作。第三项优点的重要性则体现在它能让我们更灵活地把查询结果显示出来。在默认情况下，MySQL会根据GROUP BY子句里的数据列对查询结果进行排序；可如果给查询命令增加一个ORDER BY子句，就可以让MySQL按指定顺序进行排序。比如说，如果想按人数从多到少的顺序把有总统出生的美国各州查出来并排序输出，就可以增加一个如下所示的ORDER BY子句：

```
mysql> SELECT state, COUNT(*) AS count FROM president
      -> GROUP BY state ORDER BY count DESC;
```

+-----+-----+		
state		count
+-----+-----+		
VA		8
OH		7
MA		4
NY		4
NC		2





VT		2	
TX		2	
SC		1	
NH		1	
PA		1	
KY		1	
NJ		1	
IA		1	
MO		1	
CA		1	
NE		1	
GA		1	
IL		1	
AR		1	
CT		1	
+-----+-----+			

当准备用来排序的输出列是某个统计函数的计算结果时，可以给这个输出列取一个别名，然后再把这个别名用在ORDER BY子句中。上面那个查询就是这样做的——给COUNT(\*)输出列取了一个别名叫count。另一种办法是用某输出列在整个查询结果中的出现位置来进行设定：

```
SELECT state, COUNT(*) FROM president
GROUP BY state ORDER BY 2 DESC;
```

我个人认为，用输出列的出现位置来设定排序顺序的做法存在着弊病：它很容易导致查询命令难以阅读和理解。而且，每当添加、删除或者调整了输出列的先后次序时，都不得不重新检查ORDER BY子句，以便对它们的位置编号做出相应的改变。使用别名就不存在这种问题。

类似于ORDER BY子句的情况，如果打算用GROUP BY子句对一个计算出来的输出列进行归类，可以使用输出列的别名或者它们在查询结果里的出现位置来进行设定。下面这个查询把不同月份出生的美国总统的人数分别统计了出来：

```
mysql> SELECT MONTH(birth) AS Month, MONTHNAME(birth) AS Name,
-> COUNT(*) AS count
-> FROM president GROUP BY Name ORDER BY Month;
```

+-----+-----+-----+			
Month	Name		count
+-----+-----+-----+			
1	January		4
2	February		4
3	March		4
4	April		4
5	May		2
6	June		1
7	July		4
8	August		4
9	September		1

10	October	6
11	November	5
12	December	3

这个查询可以用输出列的出现位置改写如下:

```
SELECT MONTH (birth), MONTHNAME(birth), COUNT(*)
FROM president GROUP BY 2 ORDER BY 1;
```

COUNT()函数还能与ORDER BY和LIMIT子句联合使用,比如说,如果想知道出生总统最多的前4个州是哪几个,可以对president数据表做如下查询:

```
mysql> SELECT state, COUNT(*) AS count FROM president
-> GROUP BY state ORDER BY count DESC LIMIT 4;
```

state	count
VA	8
OH	7
MA	4
NY	4

如果不想用LIMIT子句来限制查询结果中的记录个数,而只是想把与某个特定COUNT()值相对应的记录给找出来,就需要使用HAVING子句。HAVING子句与WHERE子句的相似之处是它们都是用来设定查询条件的,查询结果必须符合这些查询条件;HAVING子句与WHERE子句的不同之处是COUNT()之类的统计函数的计算结果允许在HAVING子句里出现。请看下面这个查询,它能告诉我们美国有哪些州出生了两位或两位以上的总统:

```
mysql> SELECT state, COUNT(*) AS count FROM president
-> GROUP BY state HAVING count > 1 ORDER BY count DESC;
```

state	count
VA	8
OH	7
MA	4
NY	4
NC	2
VT	2
TX	2

一般说来,带有HAVING子句的查询命令特别适合用来查找在某个数据列里重复出现的值(或者不重复出现的值——使用子句HAVING count = 1即可)。

除COUNT()以外,MySQL还为我们准备了其他一些统计函数。函数MIN()、MAX()、SUM()、AVG()能够找出或求出某个数据列里的最小值、最大值、总和、平均值。MySQL允许

把它们同时用在一个查询命令里。下面这个查询对学生们在各次考试或测验中的分数情况做了多种统计处理。查询结果中的输出列count给出了各次考试中有考试成绩的总人数（考试当天缺勤的学生不包括在内）：

```
mysql> SELECT
-> event_id,
-> MIN(score) AS minimum,
-> MAX(score) AS maximum,
-> MAX(score)-MIN(score)+1 AS range,
-> SUM(score) AS total,
-> AVG(score) AS average,
-> COUNT(score) AS count
-> FROM score
-> GROUP BY event_id;
```

event_id	minimum	maximum	range	total	average	count
1	9	20	12	439	15.1379	29
2	8	19	12	425	14.1667	30
3	60	97	38	2425	78.2258	31
4	7	20	14	379	14.0370	27
5	8	20	13	383	14.1852	27
6	62	100	39	2325	80.1724	29

很明显，如果知道event\_id列的值代表的都是哪一次考试或测验，这些信息意义就更清晰明确了。我们可以做到这一点，但需要把event数据表也包括在这个查询命令里，将在下面对这条查询命令进行改进。

MySQL提供了强大的统计功能，统计出来的信息也很有用，但也正是因为如此，所以它们也很容易被滥用。请看下面这个查询：

```
mysql> SELECT
-> state AS State,
-> AVG((TO_DAYS(death)-TO_DAYS(birth))/365) AS Age
-> FROM president WHERE death IS NOT NULL
-> GROUP BY state ORDER BY Age;
```

State	Age
KY	56.208219
VT	58.852055
NC	60.141096
OH	62.866145
NH	64.917808
NY	69.342466
NJ	71.315068



TX	71.476712	
MA	72.642009	
VA	72.822945	
PA	77.158904	
SC	78.284932	
CA	81.336986	
MO	88.693151	
IA	90.254795	
+-----+-----+		

这个查询把所有已经逝世的总统都选取出来，按他们的出生地所在州进行分组，按各州求出他们逝世时的平均年龄，再按这个平均年龄进行排序后显示出来。换句话说，这个查询能让我们了解不同州的已逝世的总统们在逝世时的平均年龄。

这又有什么意义呢？它只证明能写出这类查询命令，但并不能证明这个查询值得去编写。数据库能让我们做很多事情，但并非每件事情都有意义。当人们发现自己能够利用数据库做很多事情时，他们往往会陷入一种为查询而查询的狂热。这种对统计数字漫无目标的热衷在最近几年的体育赛事电视转播中表现得尤为明显。利用他们的数据库，赛场解说员往往能告诉你很多你确实想知道的事情，但同时也告诉你很多你根本就不想知道或者根本就没有想到过的事情。比如说，你真的关心哪支橄榄球队的三分线后卫在他所属的球队落后对手14分的情况下在赛事第二节最后两分钟里在15码线区域内阻截对手的次数最多吗？

#### 10. 从多个数据表检索信息

到目前为止，我们查询出来的信息都来自一个数据表。MySQL的能力其实远不止此。前面说过，关系数据库管理系统（RDBMS）的威力在于它们能把一种东西与另一种东西关联起来，即能把来自多个数据表的信息结合在一起以解答单个数据表不足以解答的问题。本小节将对涉及多个数据表的查询命令的编写方法进行探讨。

当打算从多个数据表选取信息的时候，需要进行一种称为“结合”（join，也叫“关联”）的操作。这种叫法来源于必须把一个数据表与另一个数据表结合起来才能得到信息查询结果。结合操作是通过把两个（或多个）数据表里的同类数据进行匹配而完成的。

下面先来看一个例子。在本章前半部分的“考试记分项目的数据表”小节里，给出了一个用来检索给定日期的考试分数的查询命令，但当时没有对它进行解释。这条查询命令实际上牵涉到了3个数据表，即需要进行一次三方结合操作。现在，分两步来对它进行说明。首先，构造一个能查出给定日期的考试分数的查询命令：

```
mysql> SELECT student_id, date, score, type
-> FROM event, score
-> WHERE date = '2002-09-23'
-> AND event.event_id = score.event_id;
```

+-----+-----+-----+-----+				
student_id	date	score	type	
+-----+-----+-----+-----+				
1	2002-09-23	15	Q	
2	2002-09-23	12	Q	



3	2002-09-23	11	Q
5	2002-09-23	13	Q
6	2002-09-23	18	Q
...			

这个查询先查出给定日期的event（考试事件）记录，再利用event记录里的event\_id（考试事件编号）把score数据表里拥有相同event\_id的考试分数都查出来。每找到一组彼此匹配的event记录和score记录，就把student\_id（学生的学号）、score（考试分数）、type（考试事件的类型）显示出来。

与以前介绍的查询命令相比，这个查询在以下两方面有着显著的不同：

- 在FROM子句里，列举了多个数据表的名字，因为要从多个数据表里检索信息：

```
FROM event, score
```

- 在WHERE子句里，给出了event数据表和score数据表的结合条件：这两个数据表里的event\_id列的值必须相互匹配：

```
WHERE ... event.event_id = score.event_id
```

请注意在提及event\_id数据列时所使用的tbl\_name.col\_name语法，这是为了让MySQL知道所提到的数据表到底是哪几个。（因为两个数据表都有event\_id数据列，所以如果不加上数据表的名字以进行区分，就会产生二义性。）但这个查询命令中的其他数据列（即date、score、type）却用不着加上数据表的名字以进行区分，因为它们只在不同的数据表里出现了一次，不可能产生二义性。

我个人喜欢在每个数据列的前面都加上数据表的名字以明确地区分它们，在今后涉及到结合操作的查询示例里，我将一直沿用这个习惯。在给每一个数据列都加上它们各自所属的数据表名字以加以区分之后，这个查询将成为如下所示的样子：

```
SELECT score.student_id, event.date, score.score, event.type
FROM event, score
WHERE event.date = '2002-09-23'
AND event.event_id = score.event_id;
```

在这一阶段，利用event数据表把日期映射到一个考试事件编号，再利用这个考试事件编号把score数据表里相匹配的考试分数找出来。这个查询的输出结果只是student\_id数据列的取值，要是能把学生们的姓名直接列出来就更清晰了。学生们的学号可以利用student数据表映射为他们的姓名，这正是第二阶段要完成的工作。score和student数据表都有student\_id数据列，两个数据表里的记录能够通过这个数据列被关联起来，利用这一事实，就能把学生们的姓名也列举出来。如下所示：

```
mysql> SELECT student.name, event.date, score.score, event.type
-> FROM event, score, student
-> WHERE event.date = '2002-09-23'
-> AND event.event_id = score.event_id
-> AND score.student_id = student.student_id;
```

name	date	score	type
Megan	2002-09-23	15	Q
Joseph	2002-09-23	12	Q
Kyle	2002-09-23	11	Q
Abby	2002-09-23	13	Q
Nathan	2002-09-23	18	Q
...			

与以前介绍的查询命令相比，这个查询在以下几方面有着显著的不同：

- 在FROM子句里，除event和score数据表外，又增加了student数据表。
- 在前一个查询里，student\_id数据列不会产生二义性，所以既可以不给它加上数据表的名字（即写成student\_id的形式），也可以给它加上数据表的名字（即写成score.student\_id的形式）。但在这个查询里，因为score和student数据表都有student\_id数据列，为了避免产生二义性，就必须分别写成score.student\_id和student.student\_id以表明它们分别来自不同的数据表。
- WHERE子句里多了一个查询条件：score数据表里的记录必须在student\_id数据列上与student数据表里的记录相匹配：

```
WHERE ... score.student_id = student.student_id
```

- 查询结果将列出学生们的姓名而不是他们的学号。（当然，如果愿意，完全可以把它们同时显示出来。）

只要对这个查询里的日期值进行替换，就能查出任何一天的考试分数、参加考试的学生名单以及考试的类型。根本用不着知道学生学号或考试事件编号之类的东西，MySQL将自动查出有关的ID编号并利用它们把需要的信息找出来。

考试记分项目中的另一项工作是统计学生们的考试缺勤情况，这一情况是以学生学号和日期的形式记录在absence数据表里的。如果想看到缺勤学生的姓名（而不仅仅是学号），就需要把absence表与student表通过student\_id列的值结合起来。下面这个查询将列出缺勤学生的姓名、学号和他们的缺勤次数：

```
mysql> SELECT student.student_id, student.name,
-> COUNT(absence.date) as absences
-> FROM student, absence
-> WHERE student.student_id = absence.student_id
-> GROUP BY student.student_id;
```

student_id	name	absences
3	Kyle	1
5	Abby	1
10	Peter	2
17	Will	1
20	Avery	1

**注意** 在这个查询命令的GROUP BY子句里，在数据列名字前面加上了数据表的名字，但就这个查询而言，这并不是必要的。这是因为：GROUP BY子句作用于输出列，而这里的查询结果只包含一个名为student\_id的列，所以MySQL知道所指的是哪一个。

如果只想知道有哪些学生没有参加考试，这个查询的查询结果将正是我们所需要的。但如果还要把这份名单交到学校办公室去，他们可能会问：“其他学生的出勤情况呢？我们想知道每一名学生的出勤情况。”学校办公室想要的是同一问题的另一种解答，即每一名学生（即使他们参加了考试）的出勤情况。但因为问法不同，这个问题的答法也就不同。

为了回答这个问题，可以用LEFT JOIN来代替普通的结合操作。LEFT JOIN将使MySQL为结合操作中第一个数据表（即名称出现在关键字LEFT JOIN左边的那个数据表）里的每一个选中的数据行生成一个输出行。只要最先列出student数据表，我们就能得到全体学生（包括那些没有在absence表里出现过的学生）的考试出勤情况。这个查询的具体写法是这样的：在FROM子句里用关键字LEFT JOIN（而不是逗号）来分隔各数据表的名字，再增加一个ON子句来指定两个数据表中的数据记录的匹配关系。如下所示：

```
mysql> SELECT student.student_id, student.name,
-> COUNT(absence.date) as absences
-> FROM student LEFT JOIN absence
-> ON student.student_id = absence.student_id
-> GROUP BY student.student_id;
```

student_id	name	absences
1	Megan	0
2	Joseph	0
3	Kyle	1
4	Katie	0
5	Abby	1
6	Nathan	0
7	Liesl	0

...

在前面的“生成统计信息”小节里，曾介绍过一个对score数据表里的数据进行各种统计分析的查询。那个查询的输出结果列出了考试事件的编号，但因为当时还不知道如何结合score数据表与event数据表才能把考试事件编号映射为考试的日期和类型，所以查询结果中就没有包括考试的日期和类型。现在，我们知道应该怎样做了。下面这个查询与前面那个差不多，但当初那个简单的考试事件编号数字现在已经被取代为相应的日期和类型了：

```
mysql> SELECT
-> event.date, event.type,
-> MIN(score.score) AS minimum,
-> MAX(score.score) AS maximum,
-> MAX(score.score)-MIN(score.score)+1 AS range,
-> SUM(score.score) AS total,
```

```

-> AVG(score.score) AS average,
-> COUNT(score.score) AS count
-> FROM score, event
-> WHERE score.event_id = event.event_id
-> GROUP BY event.date;

```

date	type	minimum	maximum	range	total	average	count
2002-09-03	Q	9	20	12	439	15.1379	29
2002-09-06	Q	8	19	12	425	14.1667	30
2002-09-09	T	60	97	38	2425	78.2258	31
2002-09-16	Q	7	20	14	379	14.0370	27
2002-09-23	Q	8	20	13	383	14.1852	27
2002-10-01	T	62	100	39	2325	80.1724	29

对涉及多个数据表的查询结果里的输出列也可以使用COUNT()和AVG()等统计函数。请看下面这个查询，它将把与考试日期和考生性别的每一种组合相对应的考生人数（即考试分数的个数）和平均分数统计出来：

```

mysql> SELECT event.date, student.sex,
-> COUNT(score.score) AS count, AVG(score.score) AS average
-> FROM event, score, student
-> WHERE event.event_id = score.event_id
-> AND score.student_id = student.student_id
-> GROUP BY event.date, student.sex;

```

date	sex	count	average
2002-09-03	F	14	14.6429
2002-09-03	M	15	15.6000
2002-09-06	F	14	14.7143
2002-09-06	M	16	13.6875
2002-09-09	F	15	77.4000
2002-09-09	M	16	79.0000
2002-09-16	F	13	15.3077
2002-09-16	M	14	12.8571
2002-09-23	F	12	14.0833
2002-09-23	M	15	14.2667
2002-10-01	F	14	77.7857
2002-10-01	M	15	82.4000

考试记分项目的另一项工作（计算每位学生的期末总成绩）也可以用一個类似的查询来完成。下面就是完成这一工作的查询命令：



```

SELECT student.student_id, student.name,
SUM(score.score) AS total, COUNT(score.score) AS n
FROM event, score, student
WHERE event.event_id = score.event_id
AND score.student_id = student.student_id
GROUP BY score.student_id
ORDER BY total;

```

结合操作并非只能作用于不同的数据表——虽然乍听起来有点奇怪，但确实能把某个数据表与它自身结合起来。比如说，如果想知道是否有两位（或者多位）总统出生于同一城市，就需要检查每位总统的出生地是否与其他总统的出生地一样。下面就是完成这一查询的命令：

```

mysql> SELECT p1.last_name, p1.first_name, p1.city, p1.state
-> FROM president AS p1, president AS p2
-> WHERE p1.city = p2.city AND p1.state = p2.state
-> AND (p1.last_name != p2.last_name OR p1.first_name != p2.first_name)
-> ORDER BY state, city, last_name;

```

last_name	first_name	city	state
Adams	John Quincy	Braintree	MA
Adams	John	Braintree	MA

这个查询命令有两个地方特别值得注意：

- 它两次用到同一个数据表（即president表），所以必须为它创建两个别名（p1和p2）才能把前、后两个president表实例中的同名数据列区别开来。
- 每位总统的记录都将与该记录本身相匹配，但这并不是我们想要的输出结果——必须确保每位总统的记录只能与其他总统的记录进行比较，WHERE子句第二行里的条件表达式就是用来剔除“记录与它本身相匹配”的情况的。

用一个类似的查询可以查出是否有两位（或者多位）总统出生在同一天。可是，如果直接用某两位总统的出生日期值进行比较，就只能把同年同月同日出生的总统查出来，那些出生在同一天但出生年份不同的总统将不会出现在查询结果里。因此，必须用MONTH()和DAYOFMONTH()函数把出生日期值中的月份和日期提取出来，如下所示：

```

mysql> SELECT p1.last_name, p1.first_name, p1.birth
-> FROM president AS p1, president AS p2
-> WHERE MONTH(p1.birth) = MONTH(p2.birth)
-> AND DAYOFMONTH(p1.birth) = DAYOFMONTH(p2.birth)
-> AND (p1.last_name != p2.last_name OR p1.first_name != p2.first_name)
-> ORDER BY p1.last_name;

```

last_name	first_name	birth
Harding	Warren G.	1865-11-02
Polk	James K.	1795-11-02

用DAYOFYEAR()来代替MONTH()和DAYOFMONTH()的组合会使我们得到一个稍微简单点儿的查询命令，但它的查询结果却可能是不正确的——因为没有考虑到闰年的因素。

到目前为止，所进行的结合操作都是在有某种逻辑联系的多个数据表上进行的，但这种所谓的逻辑联系是站在查询者的立场上说的。MySQL并不知道（也不关心）参加结合操作的数据表之间是否存在着合乎逻辑的联系。比如说，完全可以把event数据表和president数据表结合起来并查出是否在某位总统的生日那天对学生们进行了考试或测验：

```
mysql> SELECT president.last_name, president.first_name,
-> president.birth, event.type
-> FROM president, event
-> WHERE MONTH(president.birth) = MONTH(event.date)
-> AND DAYOFMONTH(president.birth) = DAYOFMONTH(event.date);
```

last_name	first_name	birth	type
Carter	James E.	1924-10-01	T

从上面的查询结果可以看出，确实在某位总统的生日那天对学生们进行了考试。但这又怎么样呢？MySQL“高高兴兴地”给出了一个“正确的”查询结果，但它并不知道（也不关心）这有什么实际的意义。这只能证明你会使用MySQL——用得还很熟练，可这并不等于说这个查询有着实际的意义和价值。不论是好是坏，计算机并不能代替人类去进行思考，想做的事仍需要由自己来决定。

#### 1.4.9 删除或更新现有的数据记录

有时候，需要删除一些现有的数据记录或者对它们的内容进行改动，这就需要用到DELETE或UPDATE语句。本小节的讨论重点就是这两条语句的用法。

DELETE语句的基本格式如下所示：

```
DELETE FROM tbl_name
WHERE which records to delete;
```

WHERE子句是可选的，用来指出哪些数据记录应该删除；如果没有WHERE子句，数据表里的全部记录将都删除。这意味着形式越简单的DELETE语句往往越危险，比如说：

```
DELETE FROM tbl_name;
```

将把数据表里的内容删除得一干二净，请千万小心！如果只想删除满足某种特定条件的数据记录，就必须用WHERE子句把它们先筛选出来。这与我们在SELECT语句里用WHERE子句来避免选取整个数据表的做法相类似。比如说，如果只想把出生在俄亥俄州的美国总统从president数据表里删除掉，就应该使用下面这样的查询：

```
mysql> DELETE FROM president WHERE state='OH';
Query OK, 7 rows affected (0.00 sec)
```

如果不清楚某条DELETE语句到底会删掉哪些数据记录，最好先把这条DELETE语句的WHERE子句放到一条SELECT语句里，看这条SELECT语句能查出哪些记录。这有助于确认那些将被删除的记录都是想要删除的——既不多也不少。比如说，如果要把Teddy Roosevelt总统的记录给删掉，那能不能使用下面这个查询呢？

```
DELETE FROM president WHERE last_name='Roosevelt';
```

这个语句确实会把Teddy Roosevelt总统的记录给删掉，但它同时还会把Franklin Roosevelt总统的记录也给删掉。因此，为保险起见，最好先把这个WHERE子句放到一条SELECT语句里检查一下，如下所示：

```
mysql> SELECT last_name, first_name FROM president
      -> WHERE last_name='Roosevelt';
```

last_name	first_name
Roosevelt	Theodore
Roosevelt	Franklin D.

从上面的查询结果可以看出，还需要对检索条件做更细致的设定：

```
mysql> SELECT last_name, first_name FROM president
      -> WHERE last_name='Roosevelt' AND first_name='Theodore';
```

last_name	first_name
Roosevelt	Theodore

现在，应该知道要用什么样的WHERE子句才能选出打算删除的记录了。下面是改正后的DELETE语句：

```
mysql> DELETE FROM president
      -> WHERE last_name='Roosevelt' AND first_name='Theodore';
```

不过，要是每删除一条数据记录都这么办，那可就太麻烦了。这的确有点麻烦，但麻烦总比后悔好！（如果遇到这种情况，可以用复制粘贴或者输入行编辑功能来尽可能地减少重复打字动作。将在第1.5节里对这方面的技巧进行介绍。）

如果想对现有记录的内容进行修改，就需要使用UPDATE语句，它的基本格式如下：

```
UPDATE tbl_name
SET which columns to change
WHERE which records to update;
```

类似于DELETE语句中的情况，这里的WHERE子句也是可选的：如果没有给出WHERE子句，就表示该数据表里的每一条记录都需要修改。比如说，下面这个查询将把每一名学生的名字都改成George：

```
mysql> UPDATE student SET name='George';
```

显然，必须谨慎对待这类查询，所以通常都必须加上一条WHERE子句来限制哪些记录需要修改。我们来看一个“美国历史研究会”场景中的例子：研究会新吸收了一名会员，但在添加他的个人资料记录项时，只填写了以下几个数据列：

```
mysql> INSERT INTO member (last_name,first_name)
-> VALUES('York','Jerome');
```

你发现自己忘了给他设置会员资格失效日期。可以用一条UPDATE语句来弥补，但需要给它加上一个适当的WHERE子句才能把想要修改的这条记录找出来：

```
mysql> UPDATE member
-> SET expiration='2001-7-20'
-> WHERE last_name='York' AND first_name='Jerome';
```

可以用一条语句对多个数据列进行修改。下面这条UPDATE语句将修改Jerome的电子邮件地址和普通邮政地址：

```
mysql> UPDATE member
-> SET email='jeromey@aol.com',street='123 Elm St',city='Anytown',
-> state='NY',zip='01003'
-> WHERE last_name='York' AND first_name='Jerome';
```

如果某个数据列允许使用NULL值，就可以把它的值设置为NULL，也就是把它设置为“无数据”状态。比如说，假如Jerome在过了一段时期之后又一次性地交齐了足以让他成为一名终身会员的会费，就应该把他的会员资格失效日期设置为NULL（意思是“永不失效”），以表明这是一位终身会员的记录：

```
mysql> UPDATE member
-> SET expiration=NULL
-> WHERE last_name='York' AND first_name='Jerome';
```

类似于DELETE语句的情况，为了确保把想要修改的记录全都筛选出来，建议最好先把UPDATE语句的WHERE子句放到一条SELECT语句里去检查一下。如果检索条件偏于严格或宽松，就会出现少修改或多修改了一些数据记录的情况。

如果读者用本小节里的查询命令做了练习，那sampdb数据库里有关的数据表就会有一些记录被删除或者被修改掉了。在开始学习下一小节之前，应该把那些改动都恢复回原状。如果真的需要重新加载那些数据表，请参考本章前半部分第1.4.7小节末尾处的指示。

## 1.5 交互式客户程序mysql的使用技巧

在本节里，将向大家介绍一些交互式客户程序mysql的使用技巧，这些技巧能帮助我们更有效率地完成任务并减少输入。此外，还将学习怎样才能更加方便快捷地连接到服务器，怎样才能不必一条一条地输入查询命令，怎样才能改变mysql程序默认的提示符（如果你不喜欢它的话），等等。



### 1.5.1 简化连接过程

在调用mysql程序的时候，通常需要设定一些连接参数，如主机名、用户名或口令等。如果在每次启动mysql程序时都不得不输入这么多的内容，你很快就会感到厌烦，也很容易出现输入错误。其实，在建立对MySQL服务器的连接的时候，有以下几种办法能减少必需的输入内容：

- 把连接参数保存在一个选项文件里。
- 利用shell的命令历史功能来重复输入有关的命令。
- 利用shell别名或脚本为mysql命令行定义一个快捷方式。

#### 1. 使用选项文件

从3.22.10版本开始，MySQL允许把连接参数保存到一个选项文件里。这样，就用不着在每次启动mysql程序的时候都输入这些参数了，系统将从选项文件里读入有关的参数，就好像已经在命令行上输入了它们一样。这样做的好处是，其他MySQL客户程序，比如mysqlimport，也能使用这些参数。换句话说，选项文件不仅能简化mysql程序的启动过程，而且也使很多其他程序的启动过程变得简单了。

在UNIX系统上，可以创建一个名为~/.my.cnf的文件（即在主目录里创建一个名为.my.cnf的文件）来作为选项文件。在Windows系统上，这个选项文件可以被创建为C盘根目录里的my.cnf文件或者Windows系统目录里的my.ini文件，即C:\my.cnf或%SYSTEM%\my.ini文件。这个选项文件是一个纯文本文件，所以可以用任何一种文本编辑器来创建它。这个文件的内容应该是下面这个样子：

```
[client]
host=server_host
user=your_name
password=your_pass
```

其中，[client]是MySQL客户程序选项组的开始标记，由此往后直到文件尾或下一个选项组开始标记之间的那些文本行将逐一给出MySQL客户程序在启动时需要用到的选项值。请把其中的server\_host、your\_name和your\_pass分别替换为在连接MySQL服务器时需要使用的主机名、用户名和口令。下面就是某个.my.cnf文件的内容：

```
[client]
host=cobra.snake.net
user=sampadm
password=secret
```

[client]是MySQL客户程序选项组的开始标记，不能省略；但那些用来定义连接参数值的文本行却是可选的——可以只把所需要的连接参数列举在选项文件里。比如说，假如使用的是UNIX系统，而MySQL用户名又与UNIX登录名一样，就不需要包括user行；假如只打算连接到本地主机，就不需要包括host行。

如果是在UNIX系统上，那么在创建了选项文件之后，还需要再多做一项工作：给这个选项文件设置限制性的访问权限，以保证别人不会读取或者修改它。下面两条命令都可以用来把这个文件的访问权限设置为只允许你本人来访问：

```
% chmod 600 .my.cnf
% chmod u=rw,go-rwx .my.cnf
```

关于选项文件的进一步讨论请参阅附录E。

## 2. 利用shell的命令历史功能

有些UNIX系统的shell（如tcsh或bash等）能把在命令行上输入过的命令记忆在一个命令历史清单里，并允许查看和重复这个清单里的命令。如果你的shell具备这一功能，就可以利用这个命令历史清单来避免敲入大段的命令内容。比如说，如果最近使用过mysql客户程序，就可以像下面这样把它从命令历史清单里找出来并重新执行一遍：

```
% !my
```

感叹号“!”的作用是：让shell从命令历史清单里把最近输入过的、以“my”打头的命令找出来并重新执行一遍，就好像在命令行上再次输入了这条命令一样。有些shell还允许利用键盘上的上、下箭头键（或Ctrl-P、Ctrl-N组合键）在命令历史清单里做前、后移动，当找到想要执行的命令后，按下回车键即可执行之。tcsh和bash具备命令历史功能，其他shell可能也具备，shell是否具备命令历史功能以及该功能的具体使用方法，可以在shell的帮助文档里查到。

## 3. 利用shell别名和脚本

如果shell允许使用别名机制，就可以建立一个短的命令名来映射到一条长命令。比如说，如果使用的shell是csh或tcsh，就可以像下面这样用alias命令来建立一个名为sampdb的别名，如下所示：

```
alias sampdb 'mysql -h cobra.snake.net -p -u sampadm sampdb'
```

如果使用的shell是bash，命令别名sampdb的定义语法稍有不同：

```
alias sampdb='mysql -h cobra.snake.net -p -u sampadm sampdb'
```

完成别名定义工作之后，下面两条命令将完全等价：

```
% sampdb
% mysql -h cobra.snake.net -p -u sampadm sampdb
```

很明显，第一个命令比第二个命令要简短得多。如果想让这个别名在每次登录进入系统的时候都能生效，就需要把它放到shell的某个启动文件（比如tcsh下的.tcshrc文件或者bash下的.bash\_profile文件）里去。

Windows系统上也存在着类似的技巧：先为mysql程序创建一个快捷方式，再通过编辑该快捷方式属性的办法把有关的连接参数包括进去。

还有一个办法能让你在调用程序或命令时减少输入：创建一个脚本。比如说，可以把mysql程序和有关的连接参数放到一个脚本文件里，再通过执行这个脚本文件来调用mysql程序。下面就是一个与上面定义的命令别名sampdb等价的shell脚本（适用于UNIX系统）：

```
#!/bin/sh
exec mysql -h cobra.snake.net -p -u sampadm sampdb
```

如果把这个脚本命名为sampdb并用chmod +x sampdb命令设置为可执行，那么在命令提示符下直接敲入sampdb就能启动mysql程序并连接到指定的数据库去。

在Windows系统上，可以利用批处理文件做同样的事情。创建一个名为sampdb.bat的批处理文件，再把下面这行文字输入到这个批处理文件里：

```
mysql -h cobra.snake.net -p -u sampadm sampdb
```

此后，执行这个批处理文件的办法有两种：一是在DOS窗口的提示符处敲入sampdb；二是双击这个批处理文件的Windows图标。

如果需要访问多个数据库或者需要连接到多个主机，那就不妨多定义几个别名、快捷方式或者脚本，让它们以不同的选项参数来调用mysql程序。

### 1.5.2 减少查询命令的输入

从对数据库进行交互式查询的角度讲，mysql是一个非常有用的程序，但它的操作界面却不适合输入冗长、多行的查询命令。虽说mysql本身并不关心所输入的查询命令是否会延续多行，但输入一条长长的查询命令却不会是件让人高兴的事。即使只是一条简短的命令，如果因为打错了字或者有语法错误而不得不再输入一遍，我想你也不会很高兴。有好几种办法能帮助我们减少不必要的录入工作：

- 利用mysql的输入行编辑器。
- 利用“复制+粘贴”机制来发出查询命令。
- 以批处理模式运行mysql程序。

#### 1. 利用mysql的输入行编辑器

mysql程序内建有GNU Readline库的输入行编辑功能，这使我们能够对输入行上的文字内容进行编辑。可以对当前输入行的内容进行编辑，也可以把以前的输入行调出来直接或经编辑之后再次输入。这非常适用于在输入的命令行里发现了打字错误并想及时纠正的场合：在按回车键之前，可以把光标移动到出错位置并改正打字错误。如果在按下回车键之后才发现自己输入的查询命令里有打字错误，可以把它调出来并在改正错误之后再次提交。（如果查询命令只有一行，改起来就更容易了。）

表1-4列出了一些输入行编辑功能的按键，这只是其中最常用的，还有很多输入行编辑命令没有收录在这个表格里。可以在bash使用手册介绍命令行编辑功能的有关章节里查到一份完整的清单，在线版bash使用手册可以在GNU Project组织的Web站点上查到，网址是：<http://www.gnu.org/manual/>。

表1-4 mysql程序的输入编辑命令

按键/组合键	含 义
上箭头键 或 Ctrl-P	调出前一个输入行
下箭头键 或 Ctrl-N	调出后一个输入行
左箭头键 或 Ctrl-B	向左移动光标
右箭头键 或 Ctrl-F	向右移动光标
Escape Ctrl-B	把光标向左移动一个单词
Escape Ctrl-F	把光标向右移动一个单词
Ctrl-A	把光标移动到输入行的开头

(续)

按键/组合键	含 义
Ctrl-E	把光标移动到输入行的末尾
Ctrl-D	删除光标位置上的那个字符
Delete	删除光标前面(左侧)的那个字符
Escape D	删除单词
Escape Backspace	删除光标前面(左侧)的那个单词
Ctrl-K	从光标位置删除到输入行的末尾
Ctrl-_	取消前一次修改;可多次重复

下面是输入行编辑功能一个简单的用法示例。假设在mysql程序里输入了如下所示的查询命令:

```
mysql> SHOW COLUMNS FROM persident;
```

在按回车键之前,注意到自己把president错误地输入成persident了。于是,先按几次左箭头键(或Ctrl-B组合键)把光标左移到字符s的位置上,再按两次Delete键以删除er,再重新输入re以改正错误,然后按下回车键以提交修改后的查询命令。如果在按下回车键之前没有发现这个打字错误也不要紧,等看到mysql显示的出错信息后,按上箭头键(或Ctrl-P组合键)调出刚刚输入的查询命令,然后再按上述过程对之进行修改就可以了。

Windows系统上的mysql程序不具备Readline编辑功能。(如果使用的是基于Windows NT的系统,mysql程序将只支持上、下箭头键的输入行查找功能和左、右箭头键的光标移动功能,但其他的编辑命令就不支持了。)要使用全套的输入行编辑命令,可以使用mysqlc程序。mysqlc程序与mysql程序的功能相同,但前者是用包含有Readline支持功能的Cygnum库编写出来的。如果想知道自己的系统上是否安装有mysqlc程序以及它是否安装得正确,请参阅附录E中对mysql程序进行的介绍。

## 2. 利用“复制+粘贴”机制来发出查询命令

如果工作在一个窗口化的操作环境里,可以通过“复制+粘贴”操作把有用的查询命令保存到一个文件里供今后使用。操作步骤如下:

- 1) 在一个终端窗口(UNIX)或一个DOS控制台窗口(Windows)里调用mysql程序。
- 2) 在一个文档窗口里打开用来存放查询命令的文件(比如说,在UNIX系统上,使用vi;在Windows系统上,使用gvim;在Mac OS系统上,使用BBEdit)。
- 3) 在文件里找到想要执行的查询命令,选取并复制下来。再切换到终端窗口或DOS控制台窗口,把刚才复制下来的查询命令粘贴到mysql程序的提示符处。

这一过程看起来比较繁琐,但熟练掌握之后却相当快捷,它能让你迅速而又不需打字地输入一条查询命令。

利用这一技巧,可以在文档窗口里先对查询命令进行编辑,也可以通过“复制+粘贴”功能利用现有查询命令的片段拼凑出一个新的查询命令来。比如说,如果需要频繁选取某特定数据表里的数据行,但又需要以不同的排序方式查看查询结果,就可以在文档里预先准备好一些彼此不同的ORDER BY子句。然后,等需要进行有关查询的时候,再根据具体情况从中挑选出一



条ORDER BY子句去拼凑出最终的查询命令来。

还可以把“复制+粘贴”操作反过来用（即把有关命令从终端窗口复制到查询命令存档文件里去）。当在mysql程序里输入查询命令的时候，它们会被保存到主目录的一个名为.mysql\_history的文件里去。如果想把自己输入的某个查询命令保存起来供今后使用，可以这样做：退出mysql程序，用一个文本编辑器打开.mysql\_history文件，找到这条查询命令并把它从.mysql\_history文件“复制+粘贴”到查询命令存档文件里去。

### 3. 以批处理模式运行mysql程序

mysql程序并非只能运行在交互模式下。mysql程序能够以非交互（即批处理）模式运行并从指定的某个文件里读取其输入。如果需要定期运行查询命令，这个技巧将特别有用，因为不需要在每次进行有关查询的时候都不得不重新敲入一大堆字符了。这种做法的好处是显而易见的：只要把有关的查询命令保存为文件一次，就可以反复多次地让mysql程序自己去读取并执行它们了。

我们来看一个“美国历史研究会”场景中的查询示例。假定经常需要通过member数据表里的interests数据列来查找哪些会员对美国历史上的某个特定事件感兴趣。比如说，为了了解哪些会员对Great Depression（美国在20世纪30年代的“大萧条”时期）感兴趣，使用下面这样的查询命令：

```
SELECT last_name, first_name, email, interests FROM member
WHERE interests LIKE '%depression%'
ORDER BY last_name, first_name;
```

把这个查询命令保存为一个名为interests.sql的文件，以后，当需要查找这些会员时，只要像下面这样把interests.sql文件馈入mysql程序里去运行就可以了：

```
% mysql sampdb < interests.sql
```

在默认情况下，以批处理模式运行的mysql程序的输出内容是以制表符来分隔的。如果想得到与以交互方式运行mysql程序时的输出格式相同的整齐效果，就需要增加一个-t选项，如下所示：

```
% mysql -t sampdb < interests.sql
```

如果想把输出结果保存起来，可以把它重定向到一个文件里去，如下所示：

```
% mysql -t sampdb < interests.sql > output_file
```

以后，当需要了解哪些会员对Thomas Jefferson总统的生平感兴趣时，只需把interests.sql文件里的depression改为Jefferson并像上面那样以批处理模式来运行mysql程序即可。不过，这个办法只有在不需要非常频繁地进行有关查询的情况下才有优势。如果必须频繁地运行某个查询命令，那就需要找一个更好的办法才行。提高查询命令灵活性的办法之一是把它保存为一个能够接受命令行参数的shell脚本，这个脚本将根据所给出的命令行参数对查询命令的具体内容做出修改，进而完成不同的查询任务。这等于是让查询命令也能接受来自命令行的参数。仍以刚才的查询命令为例，只要在运行下面这个shell脚本程序的时候给出一个兴趣值，就可以把对某特定历史事件感兴趣的会员都查出来（不妨给这个脚本起名为interests.sh）：

```

#! /bin/sh
# interests.sh - find USHL members with particular interests
if [ $# -ne 1 ]; then echo 'Please specify one keyword'; exit; fi
mysql -t sampdb <<QUERY_INPUT
SELECT last_name, first_name, email, interests FROM member
WHERE interests LIKE '%$1%'
ORDER BY last_name, first_name;
QUERY_INPUT

```

在这个脚本程序的第3行，将检查命令行参数是否有且仅有一个；如果不是这样，它就会打印一条简短的出错信息并退出执行。第4行上的<<QUERY\_INPUT到脚本程序结尾处的QUERY\_INPUT之间的文字将成为mysql程序的输入，即最终将由mysql程序去执行的查询命令。shell会把这段查询命令文本里的脚本变量\$1替换为命令行上给出的参数值（在脚本程序里，\$1、\$2等变量依次对应着该脚本的第1个、第2个命令行参数）。这样，当运行这个脚本的时候，在命令行上给出的参数值将成为查询命令中的检索关键字，而这个脚本的执行输出将正是你想要的查询结果。

在运行这个脚本程序之前，需要设置它的可执行属性，如下所示：

```
% chmod +x interests.sh
```

现在，不必在每次运行这个脚本的时候都对它进行改动了。只需通过命令行参数告诉它你想查找什么，就可以得到想要的资料：

```

% interests.sh depression
% interests.sh Jefferson

```

可以在sampdb发行版本的misc目录里找到这个interests.sh脚本，与之等价的Windows批处理文件interests.bat也可以在那儿找到。

### 1.5.3 改变mysql客户程序的提示符

从MySQL 4.0.2开始，可以根据自己的喜好去改变mysql程序的主提示符。比如说，如果想让当前数据库的名字出现在mysql程序的主提示符里，可以用下面的PROMPT命令进行修改。然后，当选取（打开）另一个数据库时，就将看到mysql程序的主提示符会发生变化：

```

% mysql
mysql> PROMPT \d>\_
PROMPT set to '\d>\_'
(none)> USE sampdb;
Database changed
sampdb> USE test;
Database changed
test>

```

关键字PROMPT的后面是打算使用的提示字符串。在这个字符串里，以反斜线字符（\）开始的序列是一些特殊的提示符选项。比如说，序列\d和\\_就分别代表着当前数据库的名字和一个空格；提示符选项的详细清单可以参见附录E中对mysql程序进行的介绍。

## 1.6 今后各章的学习计划

通过本章的学习，相信大家对MySQL的使用方法已经有了一定的了解。应该掌握的技能包括：创建数据库和数据表；对数据表里的记录进行插入、检索、修改、删除等操作。但受篇幅所限，本章提供的教程只介绍了一些最浅显的概念，还有很多内容没有涉及到。这一点可以从sampdb数据库的现状清楚地反映出来：我们创建了这个数据库并在这个数据库里创建了一些数据表，还把一些原始数据填充到了这些数据表里；在学习过程中，还编写了一些查询命令并利用它们从数据库检索出来的信息解答了一些问题；但是，仍有很多事情在等着我们去做。比如说，截止到目前，为“考试记分项目”插入新的考试分数记录和为“美国历史研究会”会员名录增加新的会员资料的工作还都不能以一种简便的交互方式来进行；现有数据记录的修改工作也只能以最原始的手段去完成；“美国历史研究会”会员名录的打印版和在线版根本就没有动手去做。这些任务需要在今后各章（尤其是第7章和第8章）的学习过程中完成。

如何开展后面的学习取决于读者对哪部分内容最感兴趣。如果你最想知道的是“美国历史研究会”和“考试记分项目”里的各项任务都是如何完成的，本书的第二部分对MySQL应用程序的设计和编写工作进行了讨论。如果你打算朝着MySQL数据库管理员的方向去努力，本书的第三部分对管理性工作进行了讨论。不过，建议大家还是先按部就班地学完第一部分、多积累一些MySQL使用方面的背景知识比较好。这些章节将帮助大家进一步了解：1) MySQL是如何对数据进行处理；2) 各种查询语句的语法和用法；3) 怎样才能让查询执行得更快。对这些内容的扎实掌握将使你胜任与MySQL有关的任何工作——无论是作为一名使用mysql程序的数据库用户、作为一名负责开发MySQL应用程序的程序员还是作为一名MySQL数据库管理员。



## 第2章 MySQL数据库里的数据

顾名思义，数据库管理系统就是用来管理数据的，所以在MySQL里做的每一件事情都或多或少地会涉及到某种形式的数据。即使是一条简单得不能再简单的SELECT 1语句，也要涉及到表达式的求值过程并且会产生一个整数类型的数据值。

MySQL里的每一个数据值都有一个类型。比如说，37.4是一个数值，'abc'是一个字符串。有时，数据的类型是很明确的。比如说，当用一条CREATE TABLE语句创建一个数据表的时候，就必须明确地为这个数据表里的每一个数据列定义一种类型：

```
CREATE TABLE mytbl
(
    int_col INT,          /* integer-valued column */
    str_col CHAR(20),     /* string-valued column */
    date_col DATE         /* date-valued column */
);
```

在另外一些场合，数据的类型就不那么明确了。比如说，当在表达式里引用一项数据、把值传递给一个函数或者使用某个函数的返回值的时候：

```
INSERT INTO mytbl (int_col,str_col,date_col)
VALUES(14,CONCAT('a','b'),20020115);
```

这条INSERT语句将完成以下几个操作，每个操作都会牵涉到有关数据类型方面的问题：

- 它把整数值14赋给整数类型的数据列int\_col。
- 它把字符串值'a'和'b'传递给CONCAT()函数。CONCAT()函数返回字符串值'ab'，而这个值又被赋给字符串类型的数据列str\_col。
- 它把整数值20020115赋给日期类型的数据列date\_col。在这个赋值操作中，出现了类型不匹配的问题。于是，MySQL自动进行了类型转换，把整数值20020115转换为日期值'2002-01-15'。

要想用好MySQL，就必须透彻理解MySQL是如何看待和处理数据的。这一章的讨论重点有两个：一是MySQL所能处理的数据值的类型；二是这些类型在实际应用中需要注意的问题：

- 包括NULL值在内，MySQL能够表达哪几大类型的值？
- MySQL为数据表里的数据列准备了哪些数据类型？每一种数据列类型（column type，也叫做“列类型”）都有哪些特性和特点？有些MySQL的列类型是很容易理解的，比如BLOB字符串类型。但有些列类型（例如，AUTO\_INCREMENT整数类型和TIMESTAMP日期类型）却有着特殊的行为特点，不明白其中奥妙的人肯定会感到吃惊。
- MySQL能够支持多少种字符集（character set）？
- 如何为数据表选择最适当的列类型？这个问题包括两个方面：1）在创建数据表的时候，如何挑选出最适用于具体情况的数据类型；2）如果有好几种类型都能用来存储数据值，它们



中的哪一种最适用?

- MySQL的表达式求值规则。为方便对数据的检索、显示和计算处理,MySQL提供了很多可以用在表达式里的操作符和函数。类型转换规则是MySQL表达式求值规则的重要组成部分之一,它指的是当表达式需要用到某个类型的值而所提供的却是另外一种类型的值时,MySQL将如何进行类型转换。而类型转换规则的要点则是类型转换会在何时发生以及到底如何进行,在某些场合,MySQL自动进行的类型转换会导致不合理或者无意义的转换结果。比如说,当把字符串'13'赋值给一个整数类型的数据列时,MySQL会自动地把它转换为整数值13;但当把字符串'abc'赋值给一个整数类型的数据列时,MySQL的转换结果却是整数值0——因为不存在与字符串'abc'对应的整数值。如果不熟悉MySQL的类型转换规则,你甚至会犯下更严重的错误并造成无法挽回的损失。比如说,原本只想修改或者删除几个数据行,可结果却把整个数据表里的数据行全都给修改或者删除掉了。

作为本章关于MySQL列类型、操作符和函数等讨论内容的补充,本书的附录B和附录C中还提供了一些额外的信息。

这一章有很多CREATE TABLE语句的使用示例。对于这条语句,相信读者不会感到陌生,因为在第1章的入门教程里已经用过多次了。如有必要,请参考附录D中对CREATE TABLE语句的介绍。本章还有几个用ALTER TABLE语句来改变数据表结构的示例,想进一步了解这条语句的读者可以参阅附录D和第3章中对它的讨论。

MySQL支持几种数据表类型(table type),不同类型的数据表有着不同的特性。在某些场合,能否使用以及如何使用某种特定数据列类型取决于该数据列所在的数据表的类型。但数据表类型并不是本章的讨论重点,它们在本章中也很少出现,对各种数据表类型的详细讨论可以参见第3章。

## 2.1 MySQL的数据类型

这里所说的“数据类型”(data type)指的是MySQL对各种数据值的基本分类。MySQL能识别和使用多种数据类型,比如数值、字符串值、日期/时间值、NULL值,等等。

### 2.1.1 数值

数值是诸如48或193.62之类的值。MySQL中的数值分整数值(没有小数部分)和浮点数值(有小数部分)两种。整数既可以表示为十进制数字,也可以表示为十六进制数字。

整数是不包含小数点的数字序列。在数值上下文里,整数可以被表示为十六进制常数,并被看做是一个64位的整数。在默认情况下,十六进制值将被当做字符串来对待,所以把十六进制值的表达语法安排在下一小节里。

浮点数由三部分组成:一个数字序列,一个小数点,再一个数字序列。其中的一个数字序列可以为空,但不能同时为空。

MySQL允许使用科学计数法。具体做法是:在整数或浮点数的后面紧跟着字母“e”或“E”,然后是一个符号(“+”或“-”),然后是一个以整数表示的指数。“1.34E+12”和“43.27e-1”

都是以合法的科学计数法表示的数值。数值“1.34E12”也是合法的——虽然它省略了指数前面的符号字符，但这只能用在MySQL 3.23.26及以后的版本里；在此前的MySQL版本里，科学计数法中的符号不允许省略。

十六进制数不能用科学计数法来表示，这是因为科学计数法的指数部分必须以字母“e”打头，而“e”又是一个合法的十六进制数字，所以用科学计数法来表示十六进制数会产生二义性。

可以在任一数值的前面加上一个负号（-）以表明它是一个负值。

### 2.1.2 字符串（字符）值

诸如'Madison, Wisconsin'或'patient shows improvement'之类的字符串也是MySQL可接受的值。字符串值两端的引号既可以是单引号，也可以是双引号。ANSI SQL标准中规定的是单引号，如果在编写查询语句时使用的是单引号，就比较容易把它们移植到其他的数据库引擎去。

MySQL能够识别出字符串里的转义序列，这些转义序列是用来表示特殊字符的，表2-1列出了一些常用的特殊字符。转义序列必须以一个反斜线字符（“\”）开始，而MySQL则会按转义规则来解释紧随其后的字符。需要提醒大家注意的是，NUL字节与NULL值是不一样的：NUL代表的是一个取值为零的字节，而NULL则代表“没有取值”的情况。

表2-1 字符串转义序列

转义序列	含 义
\0	NUL (ASCII 0)
\'	单引号
\"	双引号
\b	后退符
\n	换行符
\r	回车符
\t	制表符
\\	反斜线符
\Z	Ctrl-Z (Windows系统中的EOF字符)

如果字符串里有引号（单引号或双引号）字符，就必须用以下三种方法之一来表示：

- 如果字符串中的引号字符与字符串两端的引号相同，则双写该引号。例如：

```
'I can't'
```

```
"He said, " "I told you so.""
```

- 用与字符串中的引号字符不同的引号把该字符串引起来，此时，用不着双写字符串中的引号字符，如下所示：

```
"I can't"
```

```
'He said, "I told you so."'
```

- 用反斜杠对字符串中的引号字符进行转义，这个办法不论字符串两端的引号是哪一种都可以使用，如下所示：

```
'I can\'t'
"I can\'t"
"He said, \"I told you so.\""
'He said, \"I told you so.\"'
```

十六进制常数可以用来表示字符串值。这类常数有两种表达语法。第一种，在一个或者多个十六进制数字（0~9和a~f）前加上前缀0x，比如0x0a（十进制数值10）和0xffff（十进制数值65535）。字母形式的十六进制数字（即a~f）可以大写，也可以小写，但前缀0x不允许写成0X，也就是说，0x0a和0x0A是合法的，但0X0a和0X0A却不合法。在字符串上下文里，成对出现的十六进制数字将被解释为ASCII编码值并被转换为相应的字符，最终的结果将是一个字符串。在数值上下文里，十六进制常数将被看做是一个数字。下面的语句演示了这两种不同的用法：

```
mysql> SELECT 0x616263, 0x616263+0;
+-----+-----+
| 0x616263 | 0x616263+0 |
+-----+-----+
| abc      | 6382179    |
+-----+-----+
```

从MySQL 4.0开始，字符串值也可以用ANSI SQL表示法X'val'来表示，其中的val是成对出现的十六进制数字。类似于0x表示法，这类值通常会被解释为字符串，但也可以在数值上下文里用做数字，如下所示：

```
mysql> SELECT X'616263', X'616263'+0;
+-----+-----+
| X'616263' | X'616263'+0 |
+-----+-----+
| abc      | 6382179    |
+-----+-----+
```

与0x表示法不同的是，ANSI SQL表示法中的前缀字符“X”既可以是大写，也可以是小写：

```
mysql> SELECT X'61', x'61';
+-----+-----+
| X'61' | x'61' |
+-----+-----+
| a     | a     |
+-----+-----+
```

从MySQL 4.1开始，可以为字符串值专门指定一个字符集。在此之前，字符串值都是用服务器的默认字符集来解释的。第2.4节将对与字符集有关的问题做深入的讨论。

### 2.1.3 日期和时间值

日期和时间值指的是'2000-06-17'或'12:30:43'这样的值。MySQL允许把日期和时间合并在一起表示，比如'2000-06-17 12:30:43'。有一点请大家特别注意：MySQL是按“年-月-日”的顺序来表示日期的（这个提醒主要针对以英语为母语的人。——译者注）。虽说这是ANSI SQL标准所规定的格式（也叫做ISO 8601格式），但很多MySQL初学者还是会感到不习惯的。利用

MySQL提供的DATE\_FORMAT()函数,你可以按任意顺序来显示日期值,但“年-月-日”的顺序却是MySQL默认的数字显示格式。而且,当输入日期值时,也必须按“年-月-日”的顺序依次进行。

#### 2.1.4 NULL值

NULL值是一种不属于任何类型的值。它通常用来代表“没有数据”、“数据未知”、“数据缺失”、“数据超出取值范围”、“与本数据列无关”、“与本数据列的其他值不同”等含义。MySQL允许往数据表里插入NULL值,也允许对数据表里的NULL值进行检索或者判断某个值是否是NULL,但不允许用NULL值进行算术运算——如果你一意孤行,则计算结果将是NULL。

### 2.2 MySQL的数据列类型

数据库里的数据表是由一个或者多个数据列构成的。在用CREATE TABLE语句创建数据表的时候,必须为它的每一个数据列定义一个类型。数据列类型(column type,也叫做“列类型”)要比数据类型(data type)规定得更细致,数据类型只是一种含义宽泛的分类方法,比如“数值”或“字符串”等等。数据列类型则是对“一个给定的数据列里的值有哪些具体特性”这个问题的准确回答,比如SMALLINT或VARCHAR(32)。

MySQL中的列类型概念使我们能够只用一个简短的术语就把“某个数据列到底包含着哪一种值”的事情说清楚,同时也决定了MySQL将如何对待那些值。比如说,在MySQL里,既可以把数值类型的值保存在一个数值类型的数据列里,也可以把它们保存在一个字符串类型的数据列里,MySQL会根据为这些值选择的存储方式而区别对待这两种情况。每一种列类型都有以下几种特性(或者说可以回答以下一些问题):

- 可以把哪些种类的值保存在其中?
- 这种类型的值要占用多少存储空间,它们的长度是固定不变的(即该类型的每个值都将占用同样数量的存储空间)还是可变的(不同的值会占用不同的存储空间)?
- 这种类型的值如何进行比较和排序?
- 这种类型是否允许使用NULL值?
- 能否对这种类型进行索引?

下面先对MySQL的数据列类型进行概括性的讨论,然后再依次介绍各种数据列类型的特性。

#### 2.2.1 数据列类型概述

MySQL提供的列类型覆盖了除NULL值以外的各种数据类型分类。能否在某个数据列里使用NULL值的情况被定义为相关数据列类型的一项属性,从这个意义上讲,NULL值与各种数据列类型都有关系。

MySQL的数值类数据列类型涉及整数和浮点数两大类,如表2-2所示。整数类型的数据列既可以带正负符号,也可以不带正负符号。如果想让某个整数类型的数据列自动生成一组顺序递增的编号值,可以使用一种特殊的属性,这特别适用于需要一组独一无二的标识编号的情况。



表2-2 数值类数据列类型

数据列类型	含 义
TINYINT	非常小的整数
SMALLINT	小整数
MEDIUMINT	中等大小的整数
INT	标准整数
BIGINT	大整数
FLOAT	单精度浮点数
DOUBLE	双精度浮点数
DECIMAL	以字符串形式表示的浮点数

表2-3给出了MySQL的字符串类数据列类型。字符串可以用来容纳任何东西，包括图像和声音在内的各种二进制数据也不例外。字符串可以彼此比较，而比较操作又分为区分字母大小写和不区分字母大小写两种情况。此外，还可以对字符串进行模式匹配。（事实上，MySQL的任何一种数据列类型都允许进行模式匹配，只不过字符串类数据列类型上的模式匹配操作更常见一些而已。）

表2-3 字符串类数据列类型

数据列类型	含 义
CHAR	固定长度的字符串
VARCHAR	可变长度的字符串
TINYBLOB	非常小的BLOB（binary large object，二进制大对象）
BLOB	小BLOB
MEDIUMBLOB	中等大小的BLOB
LOB	大BLOB
TINYTEXT	非常小的文本字符串
TEXT	小文本字符串
MEDIUMTEXT	中等大小的文本字符串
LONGTEXT	大文本字符串
ENUM	枚举集合；数据列的取值将是这个枚举集合中的某一个元素
SET	集合；数据列的取值可以是这个集合中的多个元素

表2-4给出了MySQL的日期/时间类数据列类型，表中的CC、YY、MM、DD、hh、mm和ss分别代表世纪、年、月、日、时、分和秒。MySQL提供的时间类数据列类型有：日期与时间（合并表示或分开表示）、时间戳（一种专门用来记载某数据记录最近一次修改时间的列类型）。此外，为了方便不需要完整日期而只需要年份数值的情况，MySQL还准备了一个YEAR类型。

表2-4 日期/时间类数据列类型

数据列类型	含 义
DATE	日期值，格式为 'CCYY-MM-DD'
TIME	时间值，格式为 'hh:mm:ss'
DATETIME	日期加时间值，格式为 'CCYY-MM-DD hh:mm:ss'
TIMESTAMP	时间戳值，格式为 CCYYMMDDhhmmss
YEAR	年份值，格式为 CCYY

### 2.2.2 数据表的创建

在用CREATE TABLE语句创建数据表的时候，必须把构成该数据表的所有数据列都写出来。每一个数据列都有一个名字和一个类型，而每一种类型又都关联一些属性。请看下面这个示例：这条语句将创建一个名为mytbl的数据表，该数据表由三个数据列构成，而三个数据列的名字分别叫做f、c和i：

```
CREATE TABLE mytbl
(
    f FLOAT(10,4),
    c CHAR(15) NOT NULL DEFAULT 'none',
    i TINYINT UNSIGNED NULL
);
```

数据列的声明语法如下所示：

```
col_name col_type [col_attributes] [general_attributes]
```

*col\_name*是该数据列的名字，它永远出现在数据列定义的头一个。数据列的命名规则将在第3.1节里做详细介绍。简单地说，数据列的名字可以多达64个字符，允许用在数据列名字里的字符包括MySQL服务器的默认字符集中的字母和数字以及下划线（\_）和美元（\$）符号。函数名（如POS和MIN）不做保留，允许用做数据列的名字；但SQL语言关键字（如SELECT、DELETE、CREATE等）通常做保留，不能用做数据列的名字。不过，从MySQL 3.23.6开始，数据列名字允许包含字母或数字以外的其他字符，也允许把保留字用做数据列的名字，但前提是必须把这个名字用反引号（`）括起来。数据列名字的第一个字符可以是MySQL允许用在数据列名字里的任何字符——包括数字在内。不过，如果你想给数据列起一个完全由数字字符组成的名字，就必须把这个名字用反引号括起来，只有这样，才不会出现MySQL把它误认为是一个数值的情况。

*col\_type*给出了该数据列的类型，即表明这个数据列可以用来容纳什么样的数据值。此外，它还可以表明该数据列的最大长度。某些类型需要明确地设定一个长度数字，另外一些类型的长度限制则隐含在它们的名称里。比如说，CHAR(10)将把数据列的长度明确地设定为10个字符，而TINYBLOB则隐含地把数据列的最大长度设置为255个字符。有些数据列类型允许设定一个最大显示宽度（要用多少个字符来显示有关的数据值）。浮点类型还允许设定在小数点后面保留多少位数字，因此可以控制浮点数值的精确度。

在*col\_type*的后面，可以设定该类型的各种属性。这些属性的作用是对该类型做进一步的修饰和限定，会对MySQL处理这个数据列的方式产生相应的影响：

- 不同的数据列类型有着不同的独特属性。比如说，只有数值类型才有UNSIGNED属性，只有CHAR和VARCHAR类型才有BINARY属性。
- 有些属性几乎可以用来修饰任何一种数据列类型。比如说，可以用NULL或NOT NULL来限定某个数据列是否允许容纳NULL值。除BLOB和TEXT类型外，其他类型都允许利用DEFAULT *def\_value*子句来指定一个默认值*def\_value*：如果在插入新记录时没有明确地指定数据列的值，MySQL就会把这个默认值作为这个数据列的值。注意：*def\_value*的值必

须是一个常数，MySQL不允许它是一个表达式或者引用其他的数据列。

如果需要为数据列设定多项属性，它们的先后顺序就必须遵守一定的规则。一般说来，为减少不必要的麻烦，最好把某数据列类型的独特属性（比如UNSIGNED或ZEROFILL）安排在通用属性（比如NULL和NOT NULL）的前面。

下面，将依次介绍各数据列类型的声明语法和它们的独特属性（比如取值范围和存储空间要求），并用一些CREATE TABLE语句示例来演示它们的用法。方括号（[]）里的信息是可选的。比如，MEDIUMINT [(M)]里的(M)（该数据列的最大显示宽度）是可选的；VARCHAR(M)里的(M)因为没被放在方括号里，所以是必不可少的。

### 2.2.3 数值类数据列类型

MySQL的数值类数据列类型可以分为以下两类：

- 整数类型：用来存放没有小数部分的数值，如1、43、-3、0、-798 432等等。只要是能够用整数来表示的数据，比如，精确到磅的重量值、精确到英寸的长度值、家庭人口数、银河中的恒星数、培养皿中的细菌数等等，都可以用一个整数类型的数据列来保存。
- 浮点类型：用来存放可能会有小数部分的数值，如3.141 59、-0.002 73、-4.78、39.3E+4等等。涉及到小数或者那些极其巨大或微小的数值，比如，土地平均产量、价格/金额、失业率、股票价格等等，都可以用浮点类型的数据列来存放。

整数类型是最简单的。浮点类型要复杂一些，这与它们的行为在MySQL的发展史上曾多次出现重大变化有一定的关系。

可以把浮点值放到一个整数类型的数据列里去，但它将被四舍五入为一个整数；也可以把整数值放到一个浮点类型的数据列里去，它将被看做是一个小数部分等于零的浮点数。

表2-5列出了各种数值类型的名称和取值范围，表2-6列出了各种数值类型的存储空间要求。

表2-5 数值类数据列类型的取值范围

类型定义	取值范围
TINYINT [(M)]	带符号值：-128~127 ( $-2^7 \sim 2^7 - 1$ ) 不带符号值：0~255 ( $0 \sim 2^8 - 1$ )
SMALLINT [(M)]	带符号值：-32 768~32 767 ( $-2^{15} \sim 2^{15} - 1$ ) 不带符号值：0~65 535 ( $0 \sim 2^{16} - 1$ )
MEDIUMINT [(M)]	带符号值：-8 388 608~8 388 607 ( $-2^{23} \sim 2^{23} - 1$ ) 不带符号值：0~16 777 215 ( $0 \sim 2^{24} - 1$ )
INT [(M)]	带符号值：-2 147 483 648~2 147 483 647 ( $-2^{31} \sim 2^{31} - 1$ ) 不带符号值：0~4 294 967 295 ( $0 \sim 2^{32} - 1$ )
BIGINT [(M)]	带符号值：-9 223 372 036 854 775 808~9 223 372 036 854 775 807 ( $-2^{63} \sim 2^{63} - 1$ ) 不带符号值：0~18 446 744 073 709 551 615 ( $0 \sim 2^{64} - 1$ )
FLOAT [(M, D)]	最小非零值：± 1.175 494 351E-38 最大非零值：± 3.402 823 466E+38
DOUBLE [(M, D)]	最小非零值：± 2.225 073 858 507 201 4E-308 最大非零值：± 1.797 693 134 862 315 7E+308
DECIMAL [(M[,D])]	可变；取值范围由M和D的值决定

表2-6 数值类数据列类型的存储空间要求

类型定义	存储空间占用量
TINYINT [(M)]	1字节
SMALLINT [(M)]	2字节
MEDIUMINT [(M)]	3字节
INT [(M)]	4字节
BIGINT [(M)]	8字节
FLOAT [(M, D)]	4字节
DOUBLE [(M, D)]	8字节
DECIMAL [(M[,D])]	M字节 (MySQL 3.23版本以前); M+2字节 (MySQL 3.23版本及以后)

### 1. 整数类数据列类型

MySQL提供了5种整数类型，即TINYINT、SMALLINT、MEDIUMINT、INT和BIGINT。INTERGER是INT的同义词。这几种类型的取值范围各不相同，所要求的存储空间也各不相同（取值范围较大的类型需要较大的存储空间）。当给整数类型加上UNSIGNED属性以禁止负值时，该类型的取值范围将平移到从0开始的区间。

在声明一个整数列的时候，可以给它指定一个可选的显示宽度M，也就是MySQL将用多少个字符来显示该数据列里的值。M的值必须是1~255之间的某个整数。比如说，MEDIUMINT(4)定义了一个MEDIUMINT数据列，这个数据列的显示宽度是4个字符。如果没有给整数列声明一个显示宽度，MySQL将自行确定一个默认的宽度，这个默认宽度通常是该整数列里“最长”的值的长度。

需要特别注意的是，数据列里的值并不会因为设置了显示宽度而被截短为M个字符。如果某个值需要多于M个字符才能打印出来的话，MySQL会把它完整地显示出来。

整数列的显示宽度M指的是MySQL将用多少个字符来显示该数据列里的值，与整数值需要用多少个字节来存储毫不相干。比如说，不管显示宽度是多少个字符，BIGINT值都要占用8个字节的存储空间——即使把它写成BIGINT(4)，BIGINT数据列所要求的存储空间也不可能奇迹般地缩小一半。这个M与数据列的取值范围没有任何关系，可以把某个数据列声明为INT(3)，但这并不会把这个数据列里的最大值设定为999。

下面数据表创建语句能让我们了解到各种整数类数据列类型的M和D的默认值：

```
CREATE TABLE mytbl
(
    itiny      TINYINT,
    itiny_u    TINYINT UNSIGNED,
    ismall     SMALLINT,
    ismall_u   SMALLINT UNSIGNED,
    imedium    MEDIUMINT,
    imedium_u  MEDIUMINT UNSIGNED,
    ireg       INT,
    ireg_u     INT UNSIGNED,
    ibig       BIGINT,
    ibig_u     BIGINT UNSIGNED
);
```



创建出这个数据表之后，再发出一条DESCRIBE mytbl，就能从MySQL给出的输出结果中看到各有关类型的M和D的默认值了：<sup>①</sup>

Field	Type
itiny	tinyint(4)
itiny_u	tinyint(3) unsigned
ismall	smallint(6)
ismall_u	smallint(5) unsigned
imedium	mediumint(9)
imedium_u	mediumint(8) unsigned
ireg	int(11)
ireg_u	int(10) unsigned
ibig	bigint(20)
ibig_u	bigint(20) unsigned

## 2. 浮点类数据列类型

MySQL提供了3种浮点类数据列类型：FLOAT、DOUBLE和DECIMAL。DOUBLE PRECISION和RAEL是DOUBLE的同义类型，NUMERIC是DECIMAL的同义类型。浮点类型不仅有一个最大可表示值，还有一个最小非零可表示值，从这一方面讲，浮点类型的取值范围与整数类型的取值范围是不同的。浮点类型的最小非零可表示值决定了该类型的精确度，而这个精确度对科研数据的记录工作有着重要的意义。（当然，浮点类型的最大可表示值和最小非零可表示值又分正数和负数两组。）

可以将浮点类型声明为UNSIGNED属性，但MySQL 4.0.2版本之前的FLOAT和DOUBLE类型不具备这一属性。与整数类型不同，把浮点类型声明为UNSIGNED的做法不会使该类型的取值范围平移到正数区间，而只是简单地把浮点类型的负数部分消掉而已。

可以给浮点类型设定一个最大显示宽度M和一个小数保留位数D。M是一个1~255之间的整数值；D是一个0~30之间的整数值且不能大于M-2。（如果你熟悉ODBC术语，就应该看出M和D分别对应于ODBC中的“精度”和“小数位数”两个概念。）

FLOAT和DOUBLE类型的M和D值是可选的。如果省略了它们，MySQL就将按以下规则对待该类型的数据值：

- 如果MySQL的版本号小于3.23.6，FLOAT和DOUBLE类型将被看做是FLOAT(10, 2)和DOUBLE(16, 4)，它们的小数部分将分别精确到小数点后面2位和4位。
- 如果MySQL的版本号大于或等于3.23.6，FLOAT和DOUBLE类型将被保存为硬件所支持的最大精确度。

DECIMAL类型的M和D值是否可选，取决于所使用的MySQL的版本：

<sup>①</sup> 如果这条查询命令是在MySQL 3.23之前的版本上执行的，那么BIGINT数据列的默认显示宽度将是21而不是20。

- 如果MySQL的版本号小于3.23.6, DECIMAL类型的M和D值就不允许省略。
- 如果MySQL的版本号大于或等于3.23.6, 那么D的默认值将是0, M的默认值将是10。换句话说, M和D的默认值满足下面两个等价关系:

```
DECIMAL = DECIMAL(10) = DECIMAL(10,0)
DECIMAL(n) = DECIMAL(n,0)
```

为了与ODBC保持兼容, MySQL允许使用FLOAT(*p*)声明语法。但用这种语法声明的数据列有着比较复杂的行为机制:

- 如果MySQL的版本号小于3.23, *p*的取值就只能是4或8, 表示每个值将占用4个或8个字节的存储空间。FLOAT(4)和FLOAT(8)将分别等价于FLOAT(10, 2)和DOUBLE(16, 4), 即浮点值的小数部分将分别被取舍为2位和4位数字。
- 如果MySQL的版本号在3.23.0~3.23.5之间, *p*的取值仍只能是4或8, 也仍表示每个值将占用4个或8个字节的存储空间。但FLOAT(4)和FLOAT(8)将分别被看做是单精度浮点数和双精度浮点数, 而浮点值本身将被保存为硬件所支持的最大精确度。
- 如果MySQL的版本号大于或等于3.23.6, *p*的取值范围将扩大为0~53, 表示每个值将占用多少位的存储空间。如果*p*值小于24, 浮点值将被看做是单精度浮点数; 如果*p*值在25~53之间, 浮点值将被看做是双精度浮点数。

更让人糊涂的是, MySQL还允许用FLOAT4和FLOAT8来定义数据列的类型, 但它们的含义却要取决于所使用的MySQL版本:

- 如果MySQL的版本号小于3.23.6, FLOAT4和FLOAT8将分别等价于FLOAT(10, 2)和DOUBLE(16, 4)。
- 如果MySQL的版本号大于或等于3.23.6, FLOAT4和FLOAT8将分别等价于FLOAT和DOUBLE。

你可能会认为FLOAT4与FLOAT(4)以及FLOAT8与FLOAT(8)的含义是一样的, 但仔细对比一下就会发现, 它们的含义其实是有着细微差别的。

### MySQL是如何对待类型声明的

如果不知道自己的MySQL将如何对待浮点数据列的类型声明, 可以用下面的办法查出来。先创建一个数据表, 把其中的数据列声明为不确切的类型, 然后用一条DESCRIBE语句来看看MySQL报告出来的类型。比如说, 在MySQL 3.23.0里, 如果用FLOAT4创建了一个数据列, 就将看到如下所示的输出:

```
mysql> CREATE TABLE t (f FLOAT4);
mysql> DESCRIBE t;
```

Field	Type	Null	Key	Default	Extra
f	float(10,2)	YES		NULL	

在MySQL 3.23.6里，将看到如下所示的输出：

```
mysql> CREATE TABLE t (f FLOAT4);
mysql> DESCRIBE t;
```

Field	Type	Null	Key	Default	Extra
f	float	YES		NULL	

在后一种情况里，(M, D)部分被省略了，这表明浮点值将被存储为硬件所允许的最大精确度。

这个办法其实可以用来查看MySQL任何一种类型声明的实际含义，但我认为这个办法对浮点类型更有效。

### 3. 选择数值类数据列类型

在选择数值类数据列类型的时候，需要根据数据的取值范围来挑选一个取值范围最小的数据列类型来使用。选择较大的类型会浪费一部分存储空间，毫无必要地使数据表变得很大，从而降低数据的处理效率。拿整数值来说，如果数据的取值范围很小，比如人的年龄或兄弟姐妹的人数，TINYINT类型就是最佳选择。MEDIUMINT类型可以用来表示几百万个值，这对很多应用项目来说都已经是足够的了，但它在存储空间方面的开销也会大一些。BIGINT类型的取值范围更大，但它占用的存储空间将是邻近的最小的整数类型（INT）的两倍，所以应该只在必要时才使用它。再看浮点值，DOUBLE类型所占用的存储空间是FLOAT类型的两倍。因此，如果不需要非常高的精确度或者数据的取值范围不是非常大，那么使用FLOAT类型要更划算一些——这既能节约存储空间，又能提高处理效率。

数值类数据列的取值范围是由它的类型决定的。如果想把一个超出数据列取值范围的值插入到这个数据列里去，MySQL就会截短这个值：MySQL会先把这个值替换为该数据列取值范围的上限值或下限值，然后再进行插入。但MySQL在检索操作中不会对数据值进行截短处理。

数据值的截短处理是根据数据列类型的取值范围而不是它的显示宽度进行的。比如说，一个被声明为SMALLINT(3)类型的数据列的显示宽度只有3个字符，但它的取值范围却是从-32 768~32 767。如果想把值12 345插入到这个数据列里，那么，虽然12 345要比这个数据列的显示宽度更长，但仍落在该数据列的取值范围内，所以插入操作不会出现截短处理，这个值仍将以12 345的形式被插入和检索。可值99 999就不同了，它超出了这个数据列的取值范围，所以在插入时将被截短为32 767，以后的检索操作所查到的结果也将是32 767。

正常情况下，如果把一个浮点数插入到一个浮点数据列里，这个值将根据这个数据列的类型声明对小数点后面的数字进行取舍。比如说，如果把1.234 56插入到一个FLOAT(8, 1)数据列里，其结果将是1.2；如果把这个值插入到一个FLOAT(8, 4)数据列里，其结果将是1.2346。也就是说，必须根据自己对精确度的要求来为浮点数据列选定小数位数。如果数据需要精确到千分之一，就不能把小数点后面的位数声明为两位。

DECIMAL类型是一种浮点类型，但它与FLOAT和DOUBLE类型是有区别的：DECIMAL类型的值是以字符串的形式被存储起来的，它的小数点位置是固定的。这样做的好处是：DECIMAL类型的值不会像FLOAT和DOUBLE类型的数据列那样因四舍五入而产生误差，这使DECIMAL类型非常适用于财务计算。但这样做的缺点是：DECIMAL值的存储格式不同于正常的浮点值，CPU无法直接对之进行计算，所以在运算效率方面要差一些。

DECIMAL类型的最大取值范围与DOUBLE类型相同，但它的有效表示范围却要由M和D的值来决定。如果保持D值恒定而改变M值，其取值范围将随M值的增大而增大，如表2-7所示。如果保持M值恒定而改变D值，其取值范围将随D值的增大而缩小，但浮点值的精确度会增加，如表2-8所示。

表2-7 DECIMAL(M, D)类型定义中的M值对其取值范围的影响

类型定义	取值范围 (MySQL < 3.23)	取值范围 (MySQL ≥ 3.23)
DECIMAL(4, 1)	-9.9 ~ 99.9	-999.9 ~ 9 999.9
DECIMAL(5, 1)	-99.9 ~ 999.9	-9 999.9 ~ 99 999.9
DECIMAL(6, 1)	-999.9 ~ 9 999.9	-99 999.9 ~ 999 999.9

表2-8 DECIMAL(M, D)类型定义中的D值对其取值范围的影响

类型定义	取值范围 (MySQL < 3.23)	取值范围 (MySQL ≥ 3.23)
DECIMAL(4, 0)	-999 ~ 9 999	-9 999 ~ 99 999
DECIMAL(4, 1)	-9.9 ~ 99.9	-999.9 ~ 9 999.9
DECIMAL(4, 2)	-0.99 ~ 9.99	-99.99 ~ 999.99

某给定DECIMAL类型的取值范围将取决于所使用的MySQL的具体版本。从版本3.23开始，MySQL根据ANSI标准的有关规定（即一个DECIMAL(M, D)类型必须能够把任何一个总共有M个数字且小数点后面有D位数字的数值表示出来）对DECIMAL值进行处理。也就是说，一个DECIMAL(4, 2)类型必须能够把从-99.99~99.99之间的任何一个浮点值表示出来。因为正负号和小数点也要分别占用一个位置，这就会多出两个字节来，使MySQL 3.23及以后的版本中的DECIMAL(M, D)值总共要占用M+2个字节。DECIMAL(4, 2)为例，它总共需要6个字节才能把最“宽”的值（-99.99）表示出来。因为正浮点数的符号位并不一定用来存放正号字符（+），所以，作为对ANSI标准中有关规定的扩展，MySQL会用它来扩展DECIMAL(M, D)类型的取值范围。仍以DECIMAL(4, 2)为例，它能够表示的最大正浮点数将是999.99。

一般说来，一个DECIMAL(M, D)类型的值总共要占用M+2个字节。但在两种特殊的情况下，这个长度可能会缩短一点儿：

- 如果D等于0，DECIMAL值将没有小数部分，于是就不需要为小数点预留一个字节，这将使存储空间占用量减少一个字节。
- 如果某个DECIMAL数据列是UNSIGNED，就不需要为正负号预留一个字节，这也将使存储空间占用量减少一个字节。

MySQL 3.23以前的版本对DECIMAL值的表示方法与刚才的介绍有所不同。在这些早期版



本里，每个DECIMAL(M, D)值将占用M个字节的存储空间，正负号和小数点（如果有的话）也包括在这M个字节里。因此，在MySQL 3.23以前的版本里，DECIMAL(4, 2)类型的浮点值表示范围是-0.99~9.99——也就是4个字节所能表示的浮点值最大范围。如果D等于0，就不需要为小数点预留一个字节，空出来的这个字节可以多存储一个数字字符——总的效果是把数据列的取值范围扩大了一个数量级。（这就解释了表2-8里的现象：在MySQL 3.23以前的版本里，DECIMAL(4, 1)的取值范围只是DECIMAL(4, 2)的10倍；可DECIMAL(4, 0)的取值范围却是DECIMAL(4, 1)的100倍。我敢打赌：很多人根本就没有注意到这一点！）

#### 4. 数值类数据列类型的属性

ZEROFILL属性适用于所有的数值类数据列类型。如果设定了这一属性，这个数据列里的数值就会用一些前导的0来“加长”到这个数据列的显示宽度。如果想让数据列里的值全都整齐地按所设定的显示宽度显示出来，就需要使用ZEROFILL属性。准确地讲，显示宽度其实只是“最小显示宽度”，因为那些长度大于显示宽度的数值将完整地显示出来而不会被截短。这可以从下面的示例里得到证明：

```
mysql> DROP TABLE IF EXISTS mytbl;
mysql> CREATE TABLE mytbl (my_zerofill INT(5) ZEROFILL);
mysql> INSERT INTO mytbl VALUES(1),(100),(10000),(1000000);
mysql> SELECT my_zerofill FROM mytbl;
```

my_zerofill
00001
00100
10000
1000000

请注意最末尾的那个值——它的长度大于数据列的显示宽度，被完整地显示了出来。

UNSIGNED属性不允许数据列里出现负数值，它经常与各种整数类型用在一起。给一个整数数据列加上UNSIGNED属性并不会改变该数据列的取值范围的“长度”，而只是把有关数据列类型的取值范围朝正数方向平移了总长度一半的距离，使取值范围的下限值从0开始而已。请看下面这个数据表声明：

```
CREATE TABLE mytbl
(
    itiny TINYINT,
    itiny_u TINYINT UNSIGNED
);
```

itiny和itiny\_u都是TINYINT数据列，取值范围的“长度”都是256，但具体的可取值却不一样。itiny的取值范围是-128~127；而itiny\_u的取值范围却是0~255——朝正数方向平移了总长度一半（128）的距离。

如果数据不可能出现负数值，比如人口统计数字或观众人数等，就可以考虑给相应的整数数据列加上UNSIGNED属性。如果用一个带正负号的数据列类型来保存这类数据，则只能利用

上它整个取值范围的一半；而给数据列加上UNSIGNED属性之后，“活动范围”将立刻增加一倍。如果想用某个数据列来保存序列编号，给它加上UNSIGNED属性将使可用编号的数量增加一倍。

可以给浮点数据列加上UNSIGNED属性，但它的效果与整数数据列的情况稍有不同：浮点值的取值范围不会朝正数方向平移，原取值范围的正数部分将保持不变，而原取值范围的负数部分将全部变成0。需要提醒大家注意的是：如果MySQL版本低于4.0.2，就不应该给FLOAT和DOUBLE类型加上UNSIGNED属性。在较早的版本里，虽然MySQL允许把这些类型声明为UNSIGNED，但这样做的后果却往往是不可预料的数据列行为。（这条注意事项不适用于DECIMAL类型。）

另外一个属性AUTO\_INCREMENT，只能用在整数类数据列类型上。这个属性的作用是生成一组独一无二的标识符或序列编号。当把NULL值插入一个AUTO\_INCREMENT数据列的时候，MySQL会自动生成下一个序列编号并把它放到这个数据列里去。如果没有进行另外的设定，AUTO\_INCREMENT数据列的编号值将从1开始，每新增一个数据行，这个编号值就会加上一个1。如果在数据表里删除了一些数据行，这些编号值也会受到影响，具体情况取决于数据表的类型——编号值能否被复用要取决于数据表的类型。

每个数据表最多只能有一个AUTO\_INCREMENT数据列，这个数据列还必须同时具备NOT NULL属性，最好声明为PRIMARY KEY或UNIQUE键。此外，因为序列编号不可能出现负整数值，所以通常还应该给这个数据列加上UNSIGNED属性。下面这几条语句都可以用来声明一个AUTO\_INCREMENT数据列：

```
CREATE TABLE ai (i INT UNSIGNED AUTO_INCREMENT NOT NULL PRIMARY KEY);
CREATE TABLE ai (i INT UNSIGNED AUTO_INCREMENT NOT NULL, PRIMARY KEY (i));
CREATE TABLE ai (i INT UNSIGNED AUTO_INCREMENT NOT NULL, UNIQUE (i));
```

把一个AUTO\_INCREMENT数据列明确地声明为NOT NULL是一个很好的习惯，如上所示。从3.23版本开始，MySQL会自动地给AUTO\_INCREMENT数据列加上NOT NULL属性。

将在第2.3节里对AUTO\_INCREMENT数据列的行为特点做进一步的讨论。

以上介绍的各种属性都是数值类数据列所特有的。在数据列的声明定义里，在给出这些属性之后，还可以继续设定通用的NULL或NOT NULL属性。如果没有给某个数值类数据列设定NULL或NOT NULL属性，其默认属性将是NULL。还可以利用DEFAULT属性为数据列设定默认值。如果没有指定默认值，MySQL会自动地选择一个。MySQL为数值类数据列选择默认值的原则是：如果数据列里允许出现NULL值，就以NULL作为它的默认值；否则，就以0作为它的默认值。

下面这个数据表包含有3个INT数据列，它们的默认值分别是-1、1和NULL：

```
CREATE TABLE t
(
    i1 INT DEFAULT -1,
    i2 INT DEFAULT 1,
    i3 INT DEFAULT NULL
);
```

### 2.2.4 字符串类数据列类型

MySQL提供了几种字符串类的数据列类型来容纳字符数据。字符串通常用来存储和处理下面这样的值：

```
'N. Bertram, et al.'
'Pencils (no. 2 lead)'
'123 Elm St.'
'Monograph Series IX'
```

字符串可以用来表示任何一种值，从这个意义上讲，它完全有资格成为最基本的类型之一。比如说，完全可以用字符串类型来存储图像或声音之类的二进制数据；如果想把压缩数据保存起来的话，也完全可以用它来保存gzip程序的输出结果。

表2-9列出了MySQL提供的用来声明各种字符串值数据列的类型以及它们的最大长度和存储空间要求。请注意表2-9里的“L+1”、“L+2”等，这表示有关的数据列类型是一种长度可变的数据列类型。对于这些长度可变的数据列类型，有关数据值所占用的存储空间会随着不同的数据行而发生变化，数据行的总长度取决于具体存放在这些数据列里的数据值的长度。在表2-9里，用L来表示数据值的长度。

而“L+1”或“L+2”中多出来的字节是用来保存数据值的长度的。在对长度可变的数据进行处理的时候，MySQL要把数据内容和数据长度都保存起来。这些额外的字节将被当做无符号整数来对待。可变长度类型的最大长度、对应于该类型的额外字节数以及占用同样字节数的无符号整数的取值范围之间存在着一定的规律。比如说，MEDIUMBLOB值的最大长度是 $2^{24}-1$ 字节，这个长度值需要3字节来存放，而占用3字节的整数类型MEDIUMINT的最大无符号值就等于 $2^{24}-1$ ，它们之间的这种规律不是巧合。

表2-9 字符串类数据列类型

类型定义	最大长度	所需的存储空间
CHAR [(M)]	M字节	M字节
VARCHAR [(M)]	M字节	L+1字节
TINYBLOB, TINYTEXT	$2^8-1$ 字节	L+1字节
BLOB, TEXT	$2^{16}-1$ 字节	L+2字节
MEDIUMBLOB, MEDIUMTEXT	$2^{24}-1$ 字节	L+3字节
LOBLOB, LONGTEXT	$2^{32}-1$ 字节	L+4字节
ENUM ('value1', 'value2', ...)	65 535个成员	1或2字节
SET ('value1', 'value2', ...)	64个成员	1、2、3、4或8字节

ENUM和SET类型的数据列定义里都有一个列表，列表里的元素就是该数据列的合法取值。如果试图把一个没有在列表里出现的值放到这个数据列里去，它就会被转换为空字符串("")。对于其他的字符串类型，如果数据值的长度超出了它们的表示范围，MySQL将自动对有关数据进行截短处理。不过，因为字符串类型很多，取值范围从小到大一应俱全——最大的类型能够容纳将近4GB的数据，所以，从理论上讲，完全能够找到一种足够长的字符串类型来满足数据

的存储要求而不需要由MySQL去对它们进行截短处理。<sup>①</sup>

ENUM和SET类型的值在内部被保存为数值，具体细节请参阅后面“ENUM和SET数据列类型”小节的内容。其他字符串类型的值都被保存为一组连续的字节序列，并会根据它们容纳的是二进制字符串还是非二进制字符串而被区别对待为字节或者字符：

- 二进制字符串被视为一个连续排列的字节序列，与任何字符集都没有关系。MySQL将把存放在BLOB数据列以及带BINARY属性的CHAR和VARCHAR数据列里的数据当做二进制值来对待。
- 非二进制字符串被视为一个连续排列的字符序列，与具体使用的字符集有关。MySQL将把存放在TEXT数据列以及不带BINARY属性的CHAR和VARCHAR数据列里的数据当做二进制值来对待。对于单字节字符集，每个字符只占用一个字节；对于多字节字符集，每个字符要占用一个以上的字节。在MySQL 4.1及以后的版本里，不同的数据列可以使用不同的字符集；在MySQL 4.1之前的版本里，MySQL用服务器所使用的默认字符集来对字符串做出解释。

使用某种字符集将使非二进制字符串（non-binary string，即通常意义上的字符串）按照有关字符在字符集里的先后次序进行比较和排序。与此形成对照的是，二进制字符串（binary string）与具体的字符集毫无关系，也就没有什么先后次序可言。这种区别就导致了非二进制字符串与二进制字符串在解释上的差异：

- 二进制字符串的比较操作是一个字节一个字节地进行的，比较依据是两个字节里的二进制数值是否相等。这就使二进制字符串的比较操作隐含着区分大小写——因为同一个字母的大写和小写通常有着不同的数值编码。
- 非二进制字符串的比较操作是一个字符一个字符地进行的，比较依据是两个字符在字符集里的先后次序。对大多数字符集而言，同一字母的大写和小写往往有着相同的先后次序，这就使非二进制字符串的比较操作不区分大小写。此外，发音不同但写法近似的字母也往往有着相同的先后次序，比如说，如果使用的字符集是latin1，那么字符E和É就会被认为是相同的。

需要提醒大家注意的是，某些字符集的确区别对待字母的大小写情况，也会把发音不同但写法近似的字母当做不同的字符，这类字符集包括：cp1521csas、cp1527ltlcsas、latin1csas、maccecsas和macromancsas等。这些字符集名称末尾的csas代表“case sensitive, accent sensitive”（区分大小写，区分不同发音）。它们都是一些特例，因此，虽然书中其他地方讨论到非二进制字符串的时候都把它们视为不区分大小写，但读者却不应该忘记还存在着这几个不受有关规则约束的字符集特例。

字符与字节之间的区别主要体现在使用了多字节字符的字符串的长度上。请看下面这个例子：在MySQL 4.1及以后的版本里，可以通过CONVERT()函数用任何一种可用的字符集来生成

<sup>①</sup> 实际上，字符串数据列的最大有效长度取决于MySQL所使用的“客户/服务器”通信协议所支持的数据包最大长度：在MySQL 4之前的版本里，数据包的最大长度是16MB；在MySQL 4及以后的版本里，数据包的最大长度是1GB。



一个字符串。下面用ucs2字符集（这个字符集里的字符被编码为两个字节）来创建一个字符串变量@s：

```
mysql> SET @s = CONVERT('ABC' USING ucs2);
```

那么，字符串@s的“长度”到底是多少呢？这要视具体情况而定。具备多字节字符识别能力的CHAR\_LENGTH()函数将返回以字符计算的长度值，而不具备多字节字符识别能力的LENGTH()函数将返回以字节计算的长度值。因此，将得到如下所示的结果：

```
mysql> SELECT CHAR_LENGTH(@s), LENGTH(@s);
```

CHAR_LENGTH(@s)	LENGTH(@s)
3	6

二进制字符串与任何一种字符集都没有关系，它只是由一些连续排列的字节组成的一个序列而已。因此，无论是按字符计算还是按字节计算，二进制字符串的长度都是一样的，如下所示：

```
mysql> SET @s = BINARY CONVERT('ABC' USING ucs2);
```

```
mysql> SELECT CHAR_LENGTH(BINARY @s), LENGTH(BINARY @s);
```

CHAR_LENGTH(BINARY @s)	LENGTH(BINARY @s)
6	6

这种按字符计算和按字节计算得出来的长度不相同的问题对字符串数据列类型有着重要的影响。比如说，把一个数据列声明为VARCHAR(20)类型并不表示“它最多能容纳20个字符”，而是表示“它最多只能容纳可以用20个字节表示出来的字符”。对于单字节字符集，每个字符只占用一个字节，所以这两种长度将是一样的。但对于多字节字符集，它能容纳的字符个数肯定要少于20个。

### 1. CHAR和VARCHAR数据列类型

CHAR和VARCHAR是最常用的字符串类型。它们之间的区别在于CHAR是一种固定长度的类型，而VARCHAR则是一种可变长度的类型。在CHAR(M)类型的数据列里，每个值都要占用M个字节，如果某个值的长度小于M，MySQL就会在它的右边用空格字符补足。（在检索操作中，那些增补出来的空格字符将被去掉。）在VARCHAR(M)类型的数据列里，每个值只占用刚好够用的字节再加上一个用来记录其长度的字节（即总共占用L+1个字节），多余的空格会在插入操作的过程中被去掉——这与ANSI SQL标准对VARCHAR类型的规定是有差异的。（在今后的MySQL版本里，VARCHAR类型中的多余空格可能不会被去掉。）

CHAR和VARCHAR数据列的最大长度M是一个1~255之间的整数。CHAR类型中的M是可选的，它的默认值是1。从MySQL 3.23开始，CHAR(0)的用法已经合法化了。CHAR(0)的主要用途是“占位子”：在声明数据列的时候，可能不知道应该把它的宽度设置为多大，因此无法为它分配存储空间，但又不能不管它，该怎么办呢？有了CHAR(0)，这个问题就很好解决了。可

以先把它声明为CHAR(0)数据列，等以后再根据具体情况用ALTER TABLE语句来加宽这个数据列。如果允许CHAR(0)数据列使用NULL值，还可以用它来表示各种on/off开关值——因为这类数据列只有NULL或空字符串两种可取值。在数据表里，CHAR(0)数据列只占用非常少的空间，即只占用一个位。从MySQL 4.0.2开始，VARCHAR(0)的用法也合法化了，即MySQL把它当做CHAR(0)来对待。

当需要在CHAR和VARCHAR类型中做出选择的时候，请记住下面两个原则：

- 如果数据都有同样的长度，选用VARCHAR类型将多占用一些存储空间，因为数据列里的每一个值还要多用一个字节来保存其长度。反之，如果数据长短不一，选用VARCHAR类型就能节省存储空间。而一个CHAR(*n*)类型的数据列却总是要占用*n*个字节，即使它是空值或NULL值时也是如此。
- 如果数据在长度方面出入不大且已经选定要使用MyISAM或ISAM类型的数据表，那么选用CHAR类型要比选用VARCHAR类型好。对MyISAM或ISAM类型的数据表来说，固定长度的数据行要比可变长度的数据行的处理效率高。

除极少数特殊情况外，最好不要在同一个数据表里混合使用CHAR和VARCHAR类型。这是因为MySQL通常会根据具体情况把数据列从一种类型转换为另一种类型（这种做法在别的数据库系统里并不多见），下面是MySQL用来判断是否需要进行这种转换的规则：

- 在一个数据表里，如果每一个数据列的长度都是固定的，那么每一个数据行的长度也将是固定的。
- 只要数据表里有一个数据列的长度是可变的，那么各数据行的长度都将是可变的。
- 如果某个数据表里的数据行的长度是可变的，那么，为了节约存储空间，MySQL会把这个数据表里的固定长度类型的数据列转换为相应的可变长度类型。

这几条原则的含义是：只要数据表里有VARCHAR、BLOB或TEXT数据列，就不可能再有固定长度的CHAR数据列了，MySQL会默默地把它们全部转换为VARCHAR类型。下面来看一个例子，假设用下面这条语句创建了一个数据表：

```
CREATE TABLE mytbl
(
    c1 CHAR(10),
    c2 VARCHAR(10)
);
```

那么，当用DESCRIBE命令来查看这个数据表的结构时，将看到如下所示的结果：

```
mysql> DESCRIBE mytbl;
```

Field	Type	Null	Key	Default	Extra
c1	varchar(10)	YES		NULL	
c2	varchar(10)	YES		NULL	

注意，因为这个数据表里存在着一个VARCHAR类型的数据列，所以MySQL把数据列c1也

转换为VARCHAR类型了。那么，能不能用ALTER TABLE命令把c1再转换回CHAR类型呢？不能，因为MySQL还会再次把数据列c1转换为VARCHAR类型。如果想把VARCHAR数据列转换为CHAR数据列，就必须对数据表里的全体VARCHAR数据列同时进行转换，如下所示：

```
mysql> ALTER TABLE mytbl MODIFY c1 CHAR(10), MODIFY c2 CHAR(10);
mysql> DESCRIBE mytbl;
```

Field	Type	Null	Key	Default	Extra
c1	char(10)	YES		NULL	
c2	char(10)	YES		NULL	

与VARCHAR一样，BLOB或TEXT类型的数据列的长度也是可变的，但它们没有与之对应的固定长度类型。因此，如果数据表里有BLOB或TEXT类型的数据列，其中就不会再有CHAR数据列了——CHAR数据列都将被转换为VARCHAR类型。

一般说来，固定长度数据列与可变长度数据列是无法共存于同一个数据表里的。但这个结论有一个例外：长度小于4个字符的CHAR数据列不会被转换为VARCHAR类型。比如说，下面这个数据表里的CHAR数据列就不会被转换为VARCHAR类型：

```
CREATE TABLE mytbl
(
    c1 CHAR(2),
    c2 VARCHAR(10)
);
```

下面是用DESCRIBE命令查看到的这个数据表的结构：

```
mysql> DESCRIBE mytbl;
```

Field	Type	Null	Key	Default	Extra
c1	char(2)	YES		NULL	
c2	varchar(10)	YES		NULL	

为什么长度小于4个字符的CHAR数据列不会被转换为VARCHAR类型呢？因为这样做往往不划算。一般说来，可变长度的VARCHAR类型能够节省存储空间，但这是以每个值都要用一个额外的字节来记录其长度为代价的。事实上，如果数据列全都很短小，MySQL甚至会把声明为VARCHAR类型的数据列全部转换为CHAR类型。这是因为这种转换往往有利于减少存储空间的总占用量，而且，对于MyISAM或ISAM数据表，长度整齐划一的数据行还能改善数据的处理效率。下面来看一个例子，假设创建了一个下面这样的数据表：

```
CREATE TABLE mytbl
(
    c0 VARCHAR(0),
    c1 VARCHAR(1),
```

```

c2 VARCHAR(2),
c3 VARCHAR(3)
);

```

当用DESCRIBE命令来查看这个数据表的结构时，就会发现MySQL已经把所有的VARCHAR数据列悄悄地转换为CHAR类型了，如下所示：

```
mysql> DESCRIBE mytbl;
```

Field	Type	Null	Key	Default	Extra
c0	char(0)	YES		NULL	
c1	char(1)	YES		NULL	
c2	char(2)	YES		NULL	
c3	char(3)	YES		NULL	

## 2. BLOB和TEXT数据列类型

BLOB是英文“binary large object”（意思是“二进制大对象”）的缩写，其基本含义是指一个能够用来盛放任何东西的容器，而且是想让它有多大，它就有多大。MySQL里的BLOB类型其实是一个由TINYBLOB、BLOB、MEDIUMBLOB和LONGBLOB等类型组成的大家庭，除各自所能容纳的信息量有大小之分以外，在其他方面则完全相同（见表2-9）。如果想保存的信息有可能急剧膨胀到非常大的地步，或者各数据行的长短差异很大，就很适合用BLOB数据列来存放。字处理文档、图像、声音、复合型数据、新闻稿件等都是很好的例子。MySQL还有一个由TINYTEXT、TEXT、MEDIUMTEXT和LONGTEXT等类型组成的TEXT家族，它们与BLOB类型家族有很多相似之处，但TEXT类型会与某个字符集发生关联，TEXT数据列上的操作都要把字符集的因素考虑在内。（在MySQL 4.1及以后的版本里，不同的TEXT数据列可以使用不同的字符集。在MySQL 4.1之前的版本里，所有的TEXT数据列都要使用MySQL服务器的默认字符集。）BLOB类型与TEXT类型的区别与前面介绍的二进制字符串与非二进制字符串的区别是相似的。比如说，在比较和排序操作中，BLOB值是不区分大小写的，TEXT值则区分大小写。

能否在BLOB和TEXT数据列上建立索引取决于具体的数据表类型，这里的原则是：

- BDB数据表允许在BLOB和TEXT数据列上建立索引，MySQL 3.23.2和更高版本中的MyISAM数据表也允许。但在为BLOB或TEXT数据列建立索引的时候，必须设定一个前缀长度n，即只依据数据值的前n个字节来建立索引。这是为了避免因用来建立索引项的原始数据过于庞大而抵消掉索引带来的好处。不过，如果打算在TEXT数据列上建立一个FULLTEXT索引，就用不着为它设定前缀长度，这是因为基于FULLTEXT索引的检索操作将以被索引数据列里的完整内容为依据。
- ISAM、HEAP和InnoDB数据表不支持BLOB和TEXT索引。

BLOB或TEXT数据列往往会有一些特殊的要求：

- BLOB和TEXT数据列里的值在长度方面往往差异巨大，所以BLOB和TEXT数据列的删除和修改操作就很容易在数据表里产生大量碎片。这就需要定期运行OPTIMIZE TABLE命令以减少碎片和改善系统性能，有关细节请参阅第4章。



- 如果正使用着非常巨大的数据值，就可能需要对MySQL服务器进行优化调整，增加max\_allowed\_packet参数的值，有关细节请参阅第11章。对于那些可能会用到非常巨大的数据值的客户（程序），也许还需要加大它们的数据包尺寸，附录E对mysql客户程序在这方面的调整办法进行了介绍。

### 3. ENUM和SET数据列类型

ENUM和SET都是比较特殊的字符串类数据列类型，它们的取值范围是一个预先定义好的列表，构成这个列表的数据成员就是ENUM或SET数据列的合法取值。这两种类型的主要区别是：ENUM数据列里包含且只包含有一个来自合法取值列表的成员，而SET数据列里则允许包含任意多个来自合法取值列表的成员（可以为空，也可以是全体成员）。换句话说，ENUM类型的合法取值不允许同时出现，而SET类型的合法取值允许同时出现。

ENUM数据列类型定义出来的是一个枚举集合。ENUM数据列里的数据值只能是在创建数据表时为该数据列定义的合法取值列表里的某一个成员。ENUM类型的合法取值列表最多允许有65 535个成员。ENUM类型通常用来表示各种分组情况。比如说，如果把数据列定义为ENUM('N', 'Y')，那它的合法取值就只能是'N'或者'Y'中的一个。还可以用ENUM类型来表示一份调查问卷中的多重选择题的答案，或者用它来表示某种产品的颜色，如下所示：

```
employees ENUM('less than 100', '100-500', '501-1500', 'more than 1500')
color ENUM('red', 'green', 'blue', 'black')
size ENUM('S', 'M', 'L', 'XL', 'XXL')
```

再举一个例子。很多网站都会在它们的Web网页上用一组互相排斥的单选按钮向访问者提供一些选项，这些选项完全能够用MySQL数据库里的一个ENUM数据列表示出来。如果打算提供在线比萨饼订购服务，就可以用一个ENUM数据列来表示顾客订购的比萨饼的厚薄，如下所示：

```
crust ENUM('thin', 'regular', 'pan style', 'deep dish')
```

如果打算用ENUM类型来表示统计数字的分布情况，就必须在创建其枚举集合时确定好分组标准，不要让统计范围出现重叠或者空缺。比如说，如果想把医学实验中的白血球计数值的分布情况记录下来，可以像下面这样来划分统计分组：

```
wbc ENUM('0-100', '101-300', '>300')
```

医学实验的结果是一个具体的白血球计数值，这个计数值落在哪个统计分组范围内，就把代表该范围的枚举值记到wbc数据列里。但需要提醒大家注意的是，这个过程是不可逆的。仍以刚才的医学实验为例，不可能用一个代表统计范围的枚举值逆推出原始的白血球计数值。用MySQL的术语来讲，就是无法把一个表示分组情况的ENUM数据列转换为一个保存准确数值的整数数据列。（如果真的需要记录准确的白血球计数值，就应该使用一个整数数据列。）

SET类型与ENUM类型的相似之处是：在创建SET数据列的时候，同样需要为它定义一个合法取值列表。SET类型与ENUM类型的不同之处是：SET数据列允许多个合法取值同时出现，而ENUM数据列只允许一个合法取值出现。SET类型的合法取值列表最多允许有64个成员。如果数据的个数有限但它们可能同时出现，就应该考虑使用SET数据列。比如说，可以用SET类型来表示汽车的选装设备，如下所示：

```
SET('luggage rack','cruise control','air conditioning','sun roof')
```

而这个SET数据列里的数据值则会根据顾客的具体要求产生多种组合，如下所示：

```
'cruise control,sun roof'  
'luggage rack,air conditioning'  
'luggage rack,cruise control,air conditioning'  
'air conditioning'  
''
```

最末尾的那个值是一个空字符串，它表示顾客没有订购任何选装设备。空字符串也是一个合法的SET值。

SET数据列里的数据值将被表示为一个单独的字符串，如果某个数据值包含有多个合法取值，就要用逗号把这个字符串里的各合法取值分隔开。很明显，这就意味着不能用一个包含有逗号的字符串作为SET合法取值。

适合用SET类型来表示信息的例子有很多，病人的症状、来自Web页面的选择结果等都是很好的例子。以病人的症状为例，很多疾病都有一个标准的症状清单，医生们会按照这个清单来询问病人是否有那些症状，病人可能只有一种症状，也可能同时有好几种（甚至全部）症状。再来看看在线比萨饼服务，在Web页面上安排了一组复选框供顾客们用来自由选择比萨饼的馅，顾客可以只选一种馅，也可以选择好几种馅。

在为ENUM或SET数据列定义合法取值列表的时候，必须考虑到以下几个因素：

- 这个列表将决定有关数据列的合法取值，这一点刚才已经讨论过了。
- 在往ENUM或SET数据列里插入数据的时候，用不着区分字母的大小写；但当检索ENUM或SET数据列里的数据时，它们将按照给该数据列定义的合法取值列表里的字母大小写情况来显示。比如说，定义了一个ENUM('N','Y')数据列，那么，在插入操作中，完全可以使用'n'或'y'，但在检索操作中，它们却都将被显示为'N'或'Y'。但这对它们的比较或排序操作没有影响，因为ENUM和SET类型不区分字母大小写。
- 在声明某个ENUM数据列的时候，必须为它定义一个合法取值列表，而那些合法取值在这个列表里的出现次序将是这个数据列上的排序操作所使用的顺序。SET数据列上的排序操作也遵守同样的规则，但因为SET数据列允许多个合法取值同时出现，所以情况可能要复杂得多。
- 如果SET数据列里同时出现了多个合法取值，那么，在检索结果里，将按它们在该数据列的合法取值列表里的出现次序显示出来。

在创建ENUM或SET数据列的时候，需要以字符串的形式列出其合法取值列表里的各个成员。因此，把ENUM和SET划分为字符串类的数据列类型。但是，这些成员在MySQL的内部是以数值形式存储的，对它们的操作也将按数值操作来对待。这意味着ENUM和SET类型要比其他的字符串类型更有效——因为它们可以用数值操作而不是字符串操作来处理。这同时也意味着ENUM和SET值既能够用在字符串上下文里，也能够用在数值上下文里。

ENUM数据列的合法取值将按照它们在该数据列的声明定义中的先后顺序被编号，这个编号从1开始。（编号0是MySQL保留的出错代码，这个出错代码的字符串形式是一个空字符串。）

ENUM数据列占用的存储空间取决于合法取值的个数。用一个字节可以表示256个值，用两个字节就能表示65 536个值。（这恰好分别是占用一个字节的整数类型TINYINT UNSIGNED和占用两个字节的整数类型SMALLINT UNSIGNED的取值范围。）因此，ENUM数据列的合法取值列表最多允许有65 536个成员（包括编号为0的出错代码在内），所占用的存储空间是一个字节还是两个字节则要视其合法取值是否多于256个而定。因为MySQL需要为出错代码预留一个位置且每一个ENUM数据列的合法取值列表都隐含地包含这个出错成员，所以在ENUM数据列的声明定义里所能给出的合法取值的最大个数就不是65 536个而是65 535个。如果试图把一个非法取值放入ENUM数据列，MySQL就会把它替换为编号为0的出错成员。

下面这个示例大家可以亲自在mysql客户程序里试试。这个示例演示了以字符串方式和数值方式来检索ENUM数据值的操作情况（请大家注意那些合法取值的数值编号以及NULL值没有数值编号的事实）：

```
mysql> CREATE TABLE e_table (e ENUM('jane','fred','will','marcia'));
mysql> INSERT INTO e_table
    -> VALUES('jane'),('fred'),('will'),('marcia'),(''),(NULL);
mysql> SELECT e, e+0, e+1, e*3 FROM e_table;
```

e	e+0	e+1	e*3
jane	1	2	3
fred	2	3	6
will	3	4	9
marcia	4	5	12
	0	1	0
NULL	NULL	NULL	NULL

ENUM值的比较操作既可以按它们的名字来进行，也可以按它们的数值编号来进行，如下所示：

```
mysql> SELECT e FROM e_table WHERE e='will';
```

e
will

```
mysql> SELECT e FROM e_table WHERE e=3;
```

e
will

MySQL允许把空字符串定义为ENUM数据列的一个合法取值。作为合法取值的空字符串将分配到一个非零的数值编号，就像ENUM数据列其他的合法取值一样。但是，把空字符串当作一个合法取值往往会引起混乱，因为空字符串的另一个角色是充当编号为0的出错成员。在下面

的例子中，试图往ENUM数据列里插入一个非法取值'x'，这将导致MySQL插入一个出错成员。如果按字符串方式进行操作，将无法分辨空字符串究竟是代表着一个合法取值还是表示出现了一个错误；但如果按数值方式进行操作，就很清楚了：

```
mysql> CREATE TABLE t (e ENUM('a', '', 'b'));
mysql> INSERT INTO t VALUES('a'), (''), ('b'), ('x');
mysql> SELECT e, e+0 FROM t;
```

e	e+0
a	1
	2
b	3
	0

SET数据列的数值表示方法与ENUM数据列稍有不同。SET数据列的合法取值并不是按顺序进行编号的。SET数据列的每一个合法取值都对应着SET值里的一个位。第一个合法取值对应着0位，第二个合法取值对应着1位，依此类推。如果数值形式的SET值等于0，就表示它是一个空字符串。如果某个合法取值出现在了SET数据列里，与之对应的位就会被置位；如果某个合法取值没有出现在SET数据列里，与之对应的位就会被清零。一个字节对应8个合法取值，所以SET数据列占用的存储空间取决于合法取值的个数。前面已经讲过，每个SET数据列最多允许有64个合法取值。因此，如果SET数据列有1~8、9~16、17~24、25~32或者33~64个合法取值，那么它就将占用1、2、3、4或者8个字节的存储空间。

因为SET值与位有着这样的对应关系，所以SET数据列的多个合法取值才能同时出现并构成一个SET值。同时，因为位的编号与SET数据列的合法取值在该数据列的合法取值列表里的定义顺序相对应，所以字符串形式的SET值就是一个由有关合法取值按它们的定义顺序构成的字符串。

下面这个示例演示了SET数据列的字符串形式与数值形式之间的关系，把数值形式的SET值分别显示为十进制数和二进制数：

```
mysql> CREATE TABLE s_table (s SET('jane', 'fred', 'will', 'marcia'));
mysql> INSERT INTO s_table
-> VALUES('jane'), ('fred'), ('will'), ('marcia'), (''), (NULL);
mysql> SELECT s, s+0, BIN(s+0) FROM s_table;
```

s	s+0	BIN(s+0)
jane	1	1
fred	2	10
will	4	100
marcia	8	1000
	0	0
NULL	NULL	NULL



在通过插入操作对SET数据列进行赋值的时候，如果被插入的值里包含有不是该数据列合法取值的子字符串，MySQL就会把它们剔除掉，只把剩余的子串赋值给这个SET数据列。同时，在SET数据列的赋值操作中，可以按任意顺序来安排各个子串而不必照搬它们在合法取值列表里的先后顺序。不过，在涉及到SET数据列的检索操作中，各子串将按它们在合法取值列表里的先后顺序被显示出来。下面来看一个例子，假设声明了一个下面这样的SET数据列来表示几种家具：

```
SET('table','lamp','chair')
```

当把 'chair, couch, table' 赋值给这个数据列时，将会发生两件事。首先，因为 'couch' 不是该数据列的一个合法取值，所以MySQL将剔除它。其次，当在今后检索到这个值时，它将被显示为 'table, chair'。这是因为，在赋值操作中，SET值中的位是由MySQL根据它们与各合法取值的对应关系而置位或者清零的——因为不存在与 'couch' 相对应的位，所以它被剔除了。在检索操作中，所看到的检索结果其实是由MySQL顺序扫描SET值中的位并按照它们与各合法取值的对应关系而用子字符串“拼凑”出来的，这就使各子字符串按照它们在合法取值列表里的顺序依次被显示出来。这种行为导致了这样一种结果：即使把一个合法取值多次赋值给一个SET数据列，它也只会出现在检索结果里出现一次。也就是说，即使把 'lamp, lamp, lamp' 赋值给上面那个SET数据列，检索结果里也只会显示一个 'lamp'。

在MySQL里，SET数据列的值与其合法取值的这种对应关系还会导致这样一种结果：如果想用字符串来检索某个特定的SET值，就必须按适当的顺序来安排各合法取值的子字符串。仍以上面那个SET数据列为例。可以在插入操作中使用 'chair, table'，但如果在检索操作中也使用 'chair, table'，就会找不到这条记录——应该用 'table, chair' 进行查找。

ENUM和SET数据列上的排序和索引操作都是以数据列取值的内部值（即数值形式的值）为依据的，与人们惯见的顺序（比如字母表顺序）可能会不一样。比如说，虽然乍一看好像有点儿不对劲，但下面的检索结果却是正确的：

```
mysql> SELECT e FROM e_table ORDER BY e;
```

```
+-----+
| e      |
+-----+
| NULL   |
|        |
| jane   |
| fred   |
| will   |
| marcia |
+-----+
```

NULL值在经过排序的检索结果里的出现位置与具体使用的MySQL版本有关。详细情况请参阅第1章里“对查询结果进行排序”小节中的讨论。

如果数据是一个有限集合，就可以利用ENUM类型的排序操作的特点把它们按所希望的顺序进行排序和输出。具体做法是：先声明一个ENUM数据列，然后把有关数据按所希望的顺序依次写在这个ENUM数据列的合法取值列表里。假定有一个用来存放橄榄球队成员个人资料的数据表，想按这些成员的场上位置（比如按教练、助理教练、四分卫、自由中卫、接球手、得

分手等)对查询结果进行排序。于是,定义了一个ENUM数据列;并把场上位置按想要的顺序列在其合法取值列表里。这样,该ENUM数据列上的排序操作就能按所设定的顺序自动生成相应的查询结果了。

如果想让某个ENUM数据列上的排序操作按正常的字符顺序进行输出,可以先用CONCAT()函数把这个ENUM数据列里的数据值转换为非ENUM字符串,然后再进行排序,如下所示:

```
mysql> SELECT CONCAT(e) AS e_str FROM e_table ORDER BY e_str;
+-----+
| e_str |
+-----+
| NULL  |
|       |
| fred  |
| jane  |
| marcia|
| will  |
+-----+
```

CONCAT()函数并不会改变ENUM数据列里的数据值,但它会把查询结果中的ENUM值先转换为正常的字符串,从而改变它们的排序输出效果。

#### 4. 字符串类数据列类型的属性

如果给CHAR或VARCHAR类型加上BINARY属性,就会使有关数据列里的值被当做二进制字节串(即一串连续排列的字节而非一串连续排列的字符)来对待。在需要区分字母大小写情况的场合,人们经常会采取这个办法。

在MySQL 4.1及以后的版本里,还可以给CHAR、VARCHAR或者TEXT数据列设定CHARACTER SET *charset*属性,其中的*charset*必须是一个合法的字符集名称。不同的数据列允许使用不同的字符集。例如,下面这个数据表中的数据列就分别使用了latin1\_de(德语)、utf8(Unicode)和sjis(日语)字符集:

```
CREATE TABLE mytbl
(
  c1 CHAR(10) CHARACTER SET latin1_de,
  c2 VARCHAR(40) CHARACTER SET utf8,
  t MEDIUMTEXT CHARACTER SET sjis
);
```

如果所使用的MySQL版本能够支持不同的数据列使用不同的字符集,就可以在DESCRIBE命令的输出结果里看到这些信息,如下所示:

```
mysql> DESCRIBE mytbl;
+-----+-----+-----+-----+-----+-----+
| Field | Type                               | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| c1    | varchar(10) character set latin1_de | YES  |     | NULL    |       |
| c2    | varchar(40) character set utf8      | YES  |     | NULL    |       |
| t     | mediumtext character set sjis       | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+-----+
```

因为二进制字符串与任何一种字符集都没有关联, 所以CHARACTER SET属性也就不适用于各种BLOB类型以及CHAR BINARY和VARCHAR BINARY类型。ENUM或SET数据列也没有CHARACTER SET属性, 因为这两类数据在MySQL的内部都是以数值方式来表示的。<sup>①</sup>

在MySQL 4.1及以后的版本里, 如果没有被声明为BINARY, 那么, 即使没有在字符数据列的声明定义里明确地为它指定一个字符集, 也会有一个与之关联的字符集。字符集可以被关联到数据列、数据表、数据库和服务器等多个级别。在创建字符数据列的时候, MySQL将依次根据下述原则为它指定一个字符集:

- 1) 如果在数据列的声明定义里指定了一个字符集, 就使用这个字符集。
- 2) 否则, 如果在数据表的声明定义里指定了一个不同于DEFAULT的字符集, 就使用那个字符集。
- 3) 否则, 如果在数据库的声明定义里指定了一个不同于DEFAULT的字符集, 就使用那个字符集。
- 4) 否则, 使用MySQL服务器的默认字符集。

换句话说, MySQL会逐步扩大它为字符数据列搜索其关联字符集的范围, 直到找到一个得到明确定义的字符集为止, 最终被找到的字符集就将用做该字符数据列的字符集。我们知道, MySQL服务器的默认字符集总是存在的。因此, 即使在较低的级别上没有明确地指定字符集, 这一搜索过程也肯定会在到达服务器级别之后划上句号。

下面来看几个例子。假设MySQL服务器上的默认字符集是greek, 当前数据库的字符集被设定为DEFAULT。在下面这条CREATE TABLE语句里, 在数据列和数据表级别上没有指定任何字符集:

```
CREATE TABLE t (c CHAR(10));
```

因为当前数据库也没有明确地指定一个字符集, 所以MySQL将一直上溯到服务器级别才能找到一个字符集 (greek) 并把它用在数据列c上。下面用DESCRIBE命令来验证一下:

```
mysql> DESCRIBE t;
```

Field	Type	Null	Key	Default	Extra
c	char(10) character set greek	YES		NULL	

下面这条CREATE TABLE语句在数据表级别上设定了一个字符集, 所以MySQL只需上溯到这一级别就能确定与数据列c相关联的字符集应该是czech:

```
CREATE TABLE t (c CHAR(10)) CHARACTER SET czech;
```

再用DESCRIBE命令来验证一下:

<sup>①</sup> 在现时期, ENUM和SET数据的字符串值是按MySQL服务器上的默认字符集来解释的。MySQL 4.1上的研发进展有望使我们在未来的MySQL版本里或者给ENUM或SET数据列另行指定一个字符集, 或者把它们声明为BINARY——这一功能在你读到这本书的时候也许早已经实现了。

```
mysql> DESCRIBE t;
```

Field	Type	Null	Key	Default	Extra
c	char(10) character set czech	YES		NULL	

第2.4节对与字符集有关的问题做了进一步的讨论。

任何一种字符串类型都能使用通用的NULL或NOT NULL属性。如果没有进行指定，NULL就将是默认的设置。不过，把一个字符串数据列声明为NOT NULL并不意味着它不能把空字符串作为数据。空值并不等于没有值，因此，千万不要因为把一个字符串数据列声明成NOT NULL而错误地认为其中只能包含非空值。如果想让字符串值全都是非空值，就必须在编写有关语句的时候明确地写出这一限制条件。

除BLOB和TEXT类型外，其余的字符串类数据列类型都可以用DEFAULT属性设定一个默认值。如果没有为它们设定默认值，就将由MySQL为它们自动挑选一个。MySQL为字符串数据列挑选默认值的原则是：如果数据列允许使用NULL值，那它的默认值就将是NULL；如果数据列不允许使用NULL值，那它的默认值就将是空字符串——但ENUM数据列是个例外，它的默认值将是其合法取值列表中的第一个成员。（不允许使用NULL值的SET数据列的默认值其实是一个空集，只不过这个空集与空字符串等价。）

### 2.2.5 日期和时间类数据列类型

MySQL的日期和时间类数据列类型有以下几种：DATE、DATETIME、TIME、TIMESTAMP和YEAR。表2-10列出了这些类型的合法取值范围。表2-11列出了这些类型的存储空间要求。

表2-10 日期和时间类数据列类型

类型定义	取值范围
DATE	'1000-01-01' ~ '9999-12-31'
TIME	'-838:59:59' ~ '838:59:59'
DATETIME	'1000-01-01 00:00:00' ~ '9999-12-31 23:59:59'
TIMESTAMP [(M)]	19700101000000 ~ 2037年的某个时刻
YEAR [(M)]	对YEAR (4): 1901 ~ 2155; 对YEAR (2): 1970 ~ 2069

表2-11 日期和时间类数据列类型的存储空间要求

类型定义	所需的存储空间
DATE	3字节 (MySQL 3.23以前的版本是4字节)
TIME	3字节
DATETIME	8字节
TIMESTAMP	4字节
YEAR	1字节



在对日期和时间值进行修改的时候，如果插入的是一个非法的日期和时间值，MySQL就会把它替换为相应的“零值”，表2-12列出了各种日期和时间类型的“零值”。这些“零值”同时也是声明有NOT NULL属性的日期和时间数据列的默认值。

表2-12 日期和时间类型的“零值”

类型定义	零 值
DATE	'0000-00-00'
TIME	'00:00:00'
DATETIME	'0000-00-00 00:00:00'
TIMESTAMP	0000000000000000
YEAR	0000

依照ANSI SQL和ISO 8601等标准，MySQL总是把日期和时间值里的年份数字放在最前面。比如说，2004年12月3日将被表示为'2004-12-03'。但MySQL也允许在输入日期值的时候偷一点儿懒。比如说，MySQL允许只输入年份的后两位数字，它会把这个两位数转换为四个数字的年份值。再比如说，MySQL允许在输入小于10的月份和日份值时省略前导的数字0。但无论如何，都得把年份数字放在最前面，把日份数字放在最末尾。MySQL不接受'12/3/99'或者'3/12/99'之类的习惯写法。将在本章后面的“日期和时间数据列的使用”小节对MySQL的日期值解释规则做进一步的讨论。

**注意** 这里所说的时间指的是服务器所在时区的时间，MySQL不对自己返回给客户的时间值做任何时区调整。

### 1. DATE、TIME和DATETIME数据列类型

DATE、TIME和DATETIME类型分别用来存放日期值、时间值、日期和时间值的结合。它们的格式分别是'CCYY-MM-DD'、'hh:mm:ss'、和'CCYY-MM-DD hh:mm:ss'，其中CC、YY、MM、DD、hh、mm、ss分别代表世纪、年、月、日、时、分、秒。在DATETIME类型里，日期值和时间值两部分都不允许缺少或省略。要是把一个DATE值赋值给一个DATETIME数据列，MySQL会自动补足一个时间值'00:00:00'；要是把一个DATETIME值赋值给一个DATE数据列，MySQL就会去掉其中的时间值。

在MySQL里，DATETIME类型里的时间值与TIME值是有区别的。DATETIME类型里的时间值代表的是几点几分。TIME值代表的是一段逝去了的时间——这正是TIME数据列的取值范围那么大以及允许出现负值的原因。

在往数据表里插入TIME值的时候，如果输入的是一个不完整的“短”值，MySQL对它的解释可能会出乎你的预料。比如说，如果把'30'和'12:30'都插入到一个TIME数据列里，就会发现它们一个被解释为'00:00:30'，而另一个却被解释为'12:30:00'。如果输入'12:30'时想表达的真正意思是“12分30秒”，就应该把它完整地写成'00:12:30'。

### 2. TIMESTAMP数据列类型

TIMESTAMP数据列的数据表示格式是CCYYMMDDhhmmss，取值范围是从19700101000000到2037年的某个时刻。这个时间区间与UNIX时间密切相关：UNIX系统把1970年的第1天当做

“日期零点”或“纪元始点”，而这里的TIMESTAMP类型则把这一天当做自己取值范围的起点。TIMESTAMP类型的取值范围的终点对应着用4个字节表示的UNIX时间的上限，那将是2037年的某一天。<sup>①</sup>

TIMESTAMP类型的特点是能够把数据行的创建或修改时间记录下来，具体做法是：

- 如果把NULL值插入一个TIMESTAMP数据列，这个数据列就将自动取值为当前的日期和时间。
- 在创建或每次修改数据行的时候，如果没有明确地对TIMESTAMP数据列进行赋值，它就会自动取值为当前的日期和时间。不过，即使数据表里有多个TIMESTAMP数据列，也只有第一个TIMESTAMP数据列里的值会发生这种“自动取值”变化。
- 对于任何一个TIMESTAMP数据列，如果把它设置为NULL，它就会在各有关操作中自动取值为当前的时间戳值。但如果把它设置为一个确切的日期和时间值（不能是NULL），就能抑制住MySQL的时间戳机制，使TIMESTAMP数据列里的值不会自动改变。

MySQL允许为TIMESTAMP类型定义一个最大显示宽度M。表2-13列出了M的允许值范围。在对TIMESTAMP数据列进行定义的时候，如果没有为它指定M值，或者是指定的M值小于0或大于14，MySQL就视之为TIMESTAMP(14)。如果给出的M值是一个奇数，MySQL将把它视为加1得到的偶数。

表2-13 TIMESTAMP类型的显示格式

类型定义	显示格式
TIMESTAMP(14)	CCYYMMDDhhmmss
TIMESTAMP(12)	YYMMDDhhmmss
TIMESTAMP(10)	YYMMDDhhmm
TIMESTAMP(8)	CCYYMMDD
TIMESTAMP(6)	YYMMDD
TIMESTAMP(4)	YYMM
TIMESTAMP(2)	YY

TIMESTAMP数据列的显示宽度与它的存储空间占用量和其中存放的数据值没有任何关系。不论显示宽度是多少，TIMESTAMP值总是要占用4个字节，也总是要以完整的14位数字（世纪、年、月、日、时、分、秒各用两位数字）精度来参加运算。下面来看一个例子，假定声明了一个数据表并往里面插入了一些数据行，如下所示：

```
mysql> CREATE TABLE mytbl (ts TIMESTAMP(8), i INT);
mysql> INSERT INTO mytbl VALUES(20020801120000,3);
mysql> INSERT INTO mytbl VALUES(20020801120001,2);
mysql> INSERT INTO mytbl VALUES(20020801120002,1);
mysql> INSERT INTO mytbl VALUES(20020801120003,0);
mysql> SELECT * FROM mytbl ORDER BY ts, i;
```

① UNIX时间的上限肯定会随着操作系统的发展而得到扩展，TIMESTAMP类型的取值范围也会随之扩大。这个问题的根源在操作系统上，需要向系统级函数库“开刀”才能得到解决。而一旦“手术”成功，MySQL只需坐享其成就行了。

ts	i
20020801	3
20020801	2
20020801	1
20020801	0

乍看上去, SELECT语句选取出来的数据行的排列显示顺序似乎是错误的——既然前一列数据都一样, 那排序结果应该是按后一列数据从小到大的顺序排列才对, 为什么不是这样呢? 这是因为MySQL必须用TIMESTAMP数据列全部的14位数字来进行排序。在这个例子里, 因为显示宽度被定义为8, 所以那些TIMESTAMP值看起来都一样, 但它们其实是各不相同的。因此, 既然它们各不相同, 检索结果的排列顺序(正如大家看到的这样)当然要由它们来决定了。

在很多场合, 希望能够在创建一条数据记录的时候记下当时的日期和时间, 还希望这个时间值不会因以后的操作而改变, 但没有一种MySQL数据列类型能直接满足这一要求, 要想实现这一点, 比较常见的办法有两种:

- 使用一个TIMESTAMP数据列。在需要创建一条新数据记录的时候, 把这个TIMESTAMP数据列设置为NULL, 即把它初始化为当前日期和时间, 如下所示:

```
INSERT INTO tbl_name (ts_col, ...) VALUES(NULL, ...);
```

在以后对这条记录进行修改的时候, 要用这个数据列里的现有数据对它进行赋值。这种明确赋值的做法将抑制住MySQL的时间戳机制, 使TIMESTAMP数据列里的值不会自动改变, 如下所示:

```
UPDATE tbl_name SET ts_col=ts_col WHERE ... ;
```

- 使用一个DATETIME数据列。在需要创建一条新数据记录的时候, 用NOW()函数来初始化这个DATETIME数据列, 如下所示:

```
INSERT INTO tbl_name (dt_col, ...) VALUES(NOW(), ...);
```

在以后对这条记录进行修改的时候, 不要去碰这个DATETIME数据列, 如下所示:

```
UPDATE tbl_name SET ... anything BUT dt_col here ... WHERE ... ;
```

在MySQL里, 同一个数据表允许有多个TIMESTAMP数据列, 可以用它们来记录多种时间戳数据。比如说, 如果需要把创建时间和最近一次修改时间同时记录下来, 可以用两个TIMESTAMP数据列来进行记录, 一个用来记录创建时间, 另一个用来记录修改时间。不过, 千万要记住两件事: 1) 要把用来记录修改时间的TIMESTAMP数据列放在最前面, 只有这样, 它才会在创建和修改有关数据记录的时候自动取值为当时的日期和时间值; 2) 在创建一条新数据记录的时候, 要用NOW()函数来初始化用来记录创建时间的TIMESTAMP数据列, 只有这样, 它的值才能反映出当时的创建时间并在以后保持不变。

### 3. YEAR数据列类型

YEAR是一种单字节的数据列类型, 既能节约存储空间, 又能提高处理效率, 非常适用于只用到年份值的场合。在声明一个YEAR数据列的时候, 可以为它设定一个显示宽度M, 这个M只

能是4或2。如果没有在YEAR数据列的声明定义里对M值进行设定,其默认值将是4。YEAR(4)的取值范围是1901年~2155年;YEAR(2)的取值范围是1970年~2069年,但只显示最后两位数字。如果只需用到日期值里的年份数字,比如出生年份、公司创建年份等等,就应该使用YEAR类型。在各种日期和时间类型当中,YEAR类型的存储空间占用量是最小的。

TINYINT类型的存储空间占用量与YEAR类型一样(都是一个字节),但它们的取值范围却是不一样的。如果想用一个整数类型来覆盖YEAR类型所能表示的年份区间,那就至少得选用SMALLINT才行——空间占用量将翻一番。因此,如果需要表示的年份都落在YEAR类型能够表示的年份区间(1970~2069)里的话,与使用SMALLINT数据列的情况相比,使用YEAR数据列将节约一半的存储空间。使用YEAR类型的另一个好处是,MySQL能自动把两位数字的年份值转换为四位数字的年份值,转换规则见后面的“确定日期值中的年份”小节。比如说,97和14将分别被转换为1997和2014。不过,需要提醒大家注意的是,如果把数值00插入到一个YEAR(4)数据列里,转换结果将是0000而不是2000。因此,如果想让00转换为2000,就必须使用它的字符串形式'00'。

#### 4. 日期和时间类数据列类型的属性

日期和时间类数据列类型没有独有的属性,但MySQL允许给它们中的任何一个加上通用的NULL或NOT NULL属性。如果没有进行设定,MySQL将默认地给它们加上NULL属性。还可以用DEFAULT属性来设定一个默认值。如果没有给它们设置默认值,就将由MySQL为它们自动挑选一个。MySQL为日期和时间数据列挑选默认值的原则是:如果数据列允许使用NULL值,那它的默认值就将是NULL;如果数据列不允许使用NULL值,那它的默认值就将是该类型的“零值”(见表2-12)。但TIMESTAMP数据列是个例外,数据表里的第一个TIMESTAMP数据列的默认值是当前的日期和时间,其余TIMESTAMP数据列(如果有的话)的默认值是该类型的“零值”。

需要提醒大家注意的是,因为默认值必须是常数,所以MySQL不允许通过NOW()函数把“当前日期和时间”设置为DATETIME数据列的默认值。如果想得到这种效果,就只能:1)在创建新记录的时候把DATETIME数据列初始化为NOW()函数的返回值;2)使用一个TIMESTAMP数据列(如果TIMESTAMP类型的特性能满足具体项目的要求的话)。

#### 5. 日期和时间数据列的使用

MySQL能够识别和使用多种格式的日期和时间值,包括字符串形式和数值形式。表2-14列出了MySQL所支持的各种日期和时间格式。

表2-14 日期和时间类型的输入格式

类 型	可用的输入格式
DATETIME、TIMESTAMP	'CCYY-MM-DD hh:mm:ss' 'YY-MM-DD hh:mm:ss' 'CCYYMMDDhhmmss' 'YYMMDDhhmmss' CCYYMMDDhhmmss YYMMDDhhmmss



(续)

类 型	可用的输入格式
DATE	'CCYY-MM-DD' 'YY-MM-DD' 'CCYYMMDD' 'YYMMDD' CCYYMMDD YYMMDD
TIME	'hh:mm:ss' 'hhmmss' hhmmss
YEAR	'CCYY' 'YY' CCYY YY

对于那些没有世纪部分(CC)的日期和时间格式,MySQL将根据下一小节“确定日期值中的年份”里的规则来进行解释。如果使用的是那些带分隔符的字符串格式,那就用不着用“-”和“:”来分隔日期或时间值中的各个部分——MySQL允许把任何一种标点符号用做日期和时间值中的分隔符。MySQL是根据上下文而不是分隔符来解释日期和时间值的。比如说,虽然人们习惯于使用“:”来分隔时间值中的各个部分,但在需要使用日期值的场合,MySQL并不会把用“:”分隔的值解释为时间值。此外,如果使用的是带分隔符的字符串格式,小于10的月、日、小时、分、秒就用不着写成两位数字。比如说,下面这些日期和时间值都是等价的:

```
'2012-02-03 05:04:09'
'2012-2-03 05:04:09'
'2012-2-3 05:04:09'
'2012-2-3 5:04:09'
'2012-2-3 5:4:09'
'2012-2-3 5:4:9'
```

不过,对于日期和时间值里的前导零,字符串格式和数值格式的解释方法是不同的。比如说,字符串'001231'将被看做是一个6位数字的日期和时间值,并被解释为DATE类型的'2000-12-31'或DATETIME类型的'2000-12-31 00:00:00'。可是,数值001231却会被看做是1231,对它的解释就有点复杂了。在这类场合,为了避免出现预想不到的后果,还是应该使用字符串形式'001231'或完整的数值形式(如果想输入的是一个DATE值,就应该把它写成20001231;如果想输入的是一个DATETIME值,就应该把它写成20001231000000)。

一般情况下,DATE、DATETIME和TIMESTAMP类型上的赋值操作可以交叉互用,但必须清楚以下几个限制条件:

- 如果把一个DATETIME或TIMESTAMP值赋值给一个DATE数据列,其中的时间值将被丢弃。

- 如果把一个DATE值赋值给一个DATETIME或TIMESTAMP数据列，MySQL将自动补足时间零值('00:00:00')。
- 不同的日期和时间类型有着不同的取值范围。尤其是TIMESTAMP类型，它的取值范围只是从1970到2037而已。因此，如果试图把一个早于1970的DATETIME值赋值给一个TIMESTAMP数据列，就不能期待得到一个合理的结果。当然，把一个晚于2037的DATETIME值赋值给一个TIMESTAMP数据列也存在着同样的问题。

MySQL有很多用来对日期和时间值进行处理的函数。有关这些函数的详细情况请参阅附录C。

#### 6. 确定日期值中的年份

对于那些带有年份值的日期和时间类型（DATE、DATETIME、TIMESTAMP、YEAR），MySQL能够自动地把两位数字的年份值转换为四位数字的年份值。这种转换是根据以下规则进行的：

- 年份值00~69将被转换为2000~2069。
- 年份值70~99将被转换为1970~1999。

把各种两位数的年份值赋值给一个YEAR数据列再把它们检索出来，就能看到上述转换规则的实际效果了，而且还能发现一些其他东西，如下所示：

```
mysql> CREATE TABLE y_table (y YEAR);
mysql> INSERT INTO y_table VALUES(68),(69),(99),(00);
mysql> SELECT * FROM y_table;
+-----+
| y      |
+-----+
| 2068   |
| 2069   |
| 1999   |
| 0000   |
+-----+
```

注意，00被转换为0000而不是2000。这是因为数值00就等于0，而0又是YEAR类型的一个合法取值。如果插入的是一个数值0，其结果就将是0000。因此，如果想使用一个没有世纪部分的值并想得到表示2000年的结果，就必须使用字符串形式'0'或'00'。如果想让MySQL把往一个YEAR数据列里插入的值看做是字符串而不是数值，就应该使用CONCAT()函数。无论输入参数是一个字符串还是一个数值，这个函数都将统一地返回一个字符串。

需要提醒大家的是，MySQL用来把两位数字的年份值转换为四位数字的年份值的转换规则只是一种比较合理的猜测。MySQL并不知道未指定世纪的两位数字到底指的是哪一年。因此，如果MySQL的年份转换规则所产生的结果与预期不一致，那还是提供一个不可能导致歧义的四位数字年份值好了。

### MySQL是否存在“千年虫”问题

MySQL本身是不存在“千年虫”问题的，因为它内部使用的是四位数字的年份值，但这需要以所提供的年份值必须能被合理解释为先决条件。两位数字的年份值可能导致歧义性解释的根源在于人们“图省事，走捷径”的心理，与MySQL并没有什么关系。如果你愿意冒这个险，MySQL的年份转换规则会帮助你，在大部分情况下，它都能猜中你的心意。但在某些场合，你将不得不亲自输入一个四位数字的年份值。比如说，如果想把一位出生于1700年以前的美国总统的生卒日期录入到president数据表里去，就必须输入四位数字的年份值。对于这种日期和时间值会跨越多个世纪的数据列，MySQL的两位数字年份值猜测规则将肯定帮不上你什么忙。

## 2.3 序列与编号

很多应用都需要使用一些独一无二的数字来作为标识编号，这种需要会出现在很多场合：会员号、抽样编号、顾客编号、程序漏洞报告或保修凭证编号等等。

在MySQL里，这种独一无二的编号机制是通过数据列的AUTO\_INCREMENT属性而自动生成一组序列编号的办法来实现的。MySQL支持多种数据表类型（table type），不同的数据表类型对AUTO\_INCREMENT数据列的处理办法是不一样的。因此，不仅需要掌握有关AUTO\_INCREMENT机制的基本概念，还必须熟悉这种机制在各种数据表类型中的差异。为帮助大家更好地掌握这一机制并避免落入各种陷阱，本节里将对AUTO\_INCREMENT数据列的工作原理做集中的介绍。此外，还将介绍几种不使用AUTO\_INCREMENT数据列的序列编号生成办法。

MySQL 3.23以前的版本只支持ISAM数据表类型。在后来的版本里，其他几种数据表类型也逐步被添加进来——先是MyISAM和HEAP类型，然后是BDB和InnoDB类型。我们将依次讨论AUTO\_INCREMENT数据列在各种数据表类型中的行为特点。有关MySQL各种数据表处理程序（table handler）的概括性介绍可以在第3章找到。

### 2.3.1 ISAM数据表里的AUTO\_INCREMENT数据列

ISAM数据表里的AUTO\_INCREMENT数据列的行为特点是：

- 如果试图把NULL值插入到一个AUTO\_INCREMENT数据列里去，MySQL将自动生成下一个序列编号并把它插入该数据列。AUTO\_INCREMENT序列从1开始编号，第一个插入数据表的记录将被编号为1，以后插入的记录将依次被编号为2、3等等。每一个自动生成的序列编号都将比该数据列里的当前最大值多1。
- 把0插入AUTO\_INCREMENT数据列的效果与插入NULL值的效果一样。不过，MySQL并没有对此做出永久的保证，因此直接插入NULL值仍是最保险的做法。
- 如果在插入一条新记录的时候没有给某个AUTO\_INCREMENT数据列明确地设定一个值，其效果与把NULL值插入这个数据列的效果一样。

- 如果在插入一条新记录的时候给某个AUTO\_INCREMENT数据列明确地设定了一个非NULL且非零的值，将出现两种情况。其一，如果某个现有记录已经使用了这个值，就会看到一条出错信息，因为AUTO\_INCREMENT数据列里的值必须是独一无二的；其二，如果还没有别的记录使用过这个值，新记录将被插入数据表，以后再插入的新记录将从下一个编号开始递增。换句话说，如果在插入新记录时给出了一个大于当前编号值的序列编号，就可以“跳过”一些编号。

跳过一些编号的做法会导致编号序列里出现断裂带，可以利用这一点来生成一组从大于1的某个数开始的编号。我们来看一个例子。假设创建了一个ISAM数据表，其中有一个AUTO\_INCREMENT数据列，想从1000而不是1开始编号。为了达到这一目的，先得插入一条“假”记录，把它的AUTO\_INCREMENT数据列的取值设置为999。这样，以后插入的记录就会从1000开始进行编号了。等“假”记录完成使命之后，就可以删掉了。

有些读者可能会问：为什么要让序列编号从大于1的某个数开始呢？一种理由是为了使全体编号都由相同个数的数字组成。比如说，如果正在生成顾客ID编号，并且预期自己的顾客不会超过100万个，就可以让这些编号从1 000 000开始。这样，每位顾客的ID编号就都将是一个7位数字（至少在顾客人数超过9 999 999之前是这样的）。让序列从大于1的某个数开始编号的其他理由可能与技术无关。比如说，在对美国历史研究会的会员进行编号的时候，为了避免出现会员因为谁排名第一而发生争执的情况，就应该不从1开始编号。要知道，人都是有虚荣心的。如果根本就没有编号为1的会员号，谁是第一会员的争执大概也就不会发生了。

- 如果删除了AUTO\_INCREMENT数据列里的编号值最大的那条记录，那么，当再次生成一条新记录时，那个编号会被再次使用。这一行为是由ISAM数据表决定的：每一个新生成的序列编号都将比有关数据列里的当前最大值多1。同样，如果把数据表里的记录全都删掉，那么所有的编号值就都将被再次使用，新序列将重新从1开始进行编号。
- 如果用UPDATE命令把AUTO\_INCREMENT数据列里的值设置成一个正被其他记录使用着的编号，就会看到一条“键字重复”的出错信息。如果新设置的编号值大于任何一个现有编号，以后再插入的新记录将从所设置的新编号值开始继续编号。
- 如果用REPLACE命令基于AUTO\_INCREMENT数据列里的值来修改数据表里的现有记录，即AUTO\_INCREMENT数据列出现在了REPLACE命令的WHERE子句里，相应的AUTO\_INCREMENT值将不会发生变化。可如果REPLACE命令是通过其他的PRIMARY KEY或UNIQUE索引来修改现有记录的（即AUTO\_INCREMENT数据列没有出现在REPLACE命令的WHERE子句里），相应的AUTO\_INCREMENT值——如果设置其为NULL（比如没有对它进行赋值）的话——就将发生变化。
- LAST\_INSERT\_ID()函数的返回值是序列中自动生成的最后一个编号。有了这个函数，即使不知道最后生成的AUTO\_INCREMENT编号到底是多少，也可以在其他语句里引用它。LAST\_INSERT\_ID()函数只与服务器的本次会话过程中生成的AUTO\_INCREMENT值有关系，它不受其他客户（程序）的AUTO\_INCREMENT活动影响。如果在与服务器的本次会话过程中尚未生成过AUTO\_INCREMENT值，LAST\_INSERT\_ID()函数的返回值将是0。



其他数据表类型中的序列编号机制都是以ISAM数据表里的AUTO\_INCREMENT机制为基础的。在其他数据表类型里实现的序列编号机制与前面介绍的情况基本上差不多，为了进一步学习其他数据表类型里的序列编号机制，请大家务必理解上面的内容。

### 2.3.2 MyISAM数据表里的AUTO\_INCREMENT数据列

MyISAM数据表对序列编号的处理是最灵活的。源自MyISAM存储格式的新机制消除了ISAM数据表在序列编号方面的很多缺陷，主要表现在：

- 在ISAM数据表里，如果删除了编号最大的那条记录，那个编号是允许再次使用的，新创建的记录将得到刚删除的那条记录的编号，这意味着序列中的编号无法严格地做到“一旦拥有，不可更改”。换句话说，在ISAM数据表里，无法保证新记录的编号是从没被使用过的。但在MyISAM数据表里，一个自动生成的序列其编号值将严格地依次递增而不会被再次使用。如果当前的最大编号是143而又删除了包含着这个编号的记录，MySQL生成的下一个编号将是144。
- 如果没有采用前面介绍过的“假”记录办法让编号从一个较大的值开始递增，ISAM序列就只能从1开始编号。但在MyISAM数据表里，可以在CREATE TABLE语句里通过AUTO\_INCREMENT = *n*选项为序列编号明确地设定一个初始值。在下面这条语句所创建的MyISAM数据表里，名为seq的AUTO\_INCREMENT数据列将从1 000 000开始编号：

```
CREATE TABLE mytbl
(
    seq INT UNSIGNED AUTO_INCREMENT NOT NULL,
    PRIMARY KEY (seq)
) TYPE = MYISAM AUTO_INCREMENT = 1000000;
```

每个MyISAM数据表最多只能有一个AUTO\_INCREMENT数据列。因此，即使数据表里有很多个数据列（大部分数据表都会有多个数据列），出现在CREATE TABLE语句最末尾的AUTO\_INCREMENT = *n*选项也不会引起歧义。

- 可以用CREATE TABLE命令改变MyISAM数据表里当前序列计数器的编号值。比如说，如果序列的当前编号值是1000，那么下面这条语句就将使下一个编号从2000开始：

```
ALTER TABLE mytbl AUTO_INCREMENT = 2000;
```

如果刚刚删除了编号最大的那条记录，并且确实想再次使用那个编号，那完全可以做到。具体做法是：把编号计数器尽可能地往低处设置，而这将使下一个编号只比现有编号的最大值多1。如下所示：

```
ALTER TABLE mytbl AUTO_INCREMENT = 1;
```

除消除了ISAM序列处理机制的上述几处缺陷外，从MySQL 3.23.5开始，MySQL的数据表处理程序（table handler）还增添了使用复合（多数据列）索引在同一数据表里创建多个相互独立的序列的能力。这项功能的具体用法是这样的：为数据表创建一个由多个数据列组成的PRIMARY KEY或UNIQUE索引，并把一个AUTO\_INCREMENT数据列包括在这个索引里作为它的最后一个数据列。这样，在这个复合索引里，前面的那些数据列每构成一种独一无二的组

合，最末尾的那个AUTO\_INCREMENT数据列就会生成一个与该组合相对应的序列编号，对应于不同组合的序列彼此互不干扰。比如说，我们想用一個名为bugs的数据表来同时记录多个软件项目的程序漏洞报告，下面是这个数据表的声明定义：

```
CREATE TABLE bugs
(
    proj_name    VARCHAR(20) NOT NULL,
    bug_id       INT UNSIGNED AUTO_INCREMENT NOT NULL,
    description  VARCHAR(100),
    PRIMARY KEY (proj_name, bug_id)
) TYPE = MYISAM;
```

其中，proj\_name数据列用来存放软件项目的名称，description数据列用来存放对程序漏洞的描述。bug\_id数据列用来存放程序漏洞的编号，它是一个AUTO\_INCREMENT数据列。我们还创建了一个与proj\_name数据列相关联的复合索引，这样，就能为不同的软件项目分别生成一个互不干扰的序列编号了。接下来，把下面5条记录插入到数据表里，其中有3条记录是SuperBrowser项目的程序漏洞，有2条记录是SpamSquisher项目的程序漏洞：

```
mysql> INSERT INTO bugs (proj_name,description)
-> VALUES('SuperBrowser','crashes when displaying complex tables');
mysql> INSERT INTO bugs (proj_name,description)
-> VALUES('SuperBrowser','image scaling does not work');
mysql> INSERT INTO bugs (proj_name,description)
-> VALUES('SpamSquisher','fails to block known blacklisted domains');
mysql> INSERT INTO bugs (proj_name,description)
-> VALUES('SpamSquisher','fails to respect whitelist addresses');
mysql> INSERT INTO bugs (proj_name,description)
-> VALUES('SuperBrowser','background patterns not displayed');
```

下面是我们发出的查询和得到的结果：

```
mysql> SELECT * FROM bugs ORDER BY proj_name, bug_id;
+-----+-----+-----+
| proj_name | bug_id | description |
+-----+-----+-----+
| SpamSquisher | 1 | fails to block known blacklisted domains |
| SpamSquisher | 2 | fails to respect whitelist addresses |
| SuperBrowser | 1 | crashes when displaying complex tables |
| SuperBrowser | 2 | image scaling does not work |
| SuperBrowser | 3 | background patterns not displayed |
+-----+-----+-----+
```

只要程序漏洞记录项是按软件项目分组排列的，哪一组在先哪一组在后无关紧要。请大家把注意力集中在bug\_id列上：程序漏洞的编号是按软件项目分别编号的。

如果用一个复合索引创建出了多个编号序列，那么，当删除了序列编号为最大值的那条记录之后又插入一条新的记录时，那个编号就会被再次使用。这与MyISAM数据表不会再次使用一个已删除编号的正常行为是对立的，所以请大家务必要注意这一点。

### 2.3.3 HEAP数据表里的AUTO\_INCREMENT数据列

HEAP数据表是从MySQL 4.1开始才允许使用AUTO\_INCREMENT数据列的，在此之前，HEAP数据表根本就不支持AUTO\_INCREMENT机制。HEAP数据表里的AUTO\_INCREMENT数据列的行为特点是：

- 序列编号的初始值既允许通过CREATE TABLE语句的AUTO\_INCREMENT = *n*选项进行设置，也允许在数据表被创建出来之后通过ALTER TABLE语句的AUTO\_INCREMENT = *n*选项加以改变。
- 如果删除了编号值最大的那条记录，它用过的编号将不允许再次使用。
- HEAP数据表不支持在一个数据表里使用复合索引来生成多个互不干扰的序列编号。

### 2.3.4 BDB数据表里的AUTO\_INCREMENT数据列

BDB数据表处理程序对AUTO\_INCREMENT数据列的处理方式是：

- 序列编号的初始值不允许通过CREATE TABLE语句的AUTO\_INCREMENT = *n*选项进行设置，也不允许在数据表被创建出来之后通过ALTER TABLE语句的AUTO\_INCREMENT = *n*选项加以改变。
- 如果删除了编号值最大的那条记录，它用过的编号允许再次使用。
- BDB数据表支持在一个数据表里使用复合索引来生成多个互不干扰的序列编号。

### 2.3.5 InnoDB数据表里的AUTO\_INCREMENT数据列

InnoDB数据表处理程序对AUTO\_INCREMENT数据列的处理方式是：

- 序列编号的初始值不允许通过CREATE TABLE语句的AUTO\_INCREMENT = *n*选项进行设置，也不允许在数据表被创建出来之后通过ALTER TABLE语句的AUTO\_INCREMENT = *n*选项加以改变。
- 如果删除了编号值最大的那条记录，它用过的编号将不允许再次使用。
- InnoDB数据表不支持在一个数据表里使用复合索引来生成多个互不干扰的序列编号。

### 2.3.6 使用AUTO\_INCREMENT机制时的注意事项

在使用AUTO\_INCREMENT数据列的时候，为避免不必要的麻烦，请记住以下几个要点：

- AUTO\_INCREMENT并不是一种数据列类型，它只是数据列类型的一个属性。进一步讲，AUTO\_INCREMENT是一种只适用于整数类型的属性。MySQL 3.23之前的版本对这一点要求得并不是很严格。它们允许给诸如CHAR这样的类型也声明上AUTO\_INCREMENT属性，但只有整数类型的AUTO\_INCREMENT数据列才能正确地工作。
- MySQL提供AUTO\_INCREMENT机制的主要目的是为了生成一个正整数序列，所以你应该把AUTO\_INCREMENT数据列声明为UNSIGNED。这种做法的另一个好处是能使序列中的编号个数增加一倍。

在某些情况下，可以利用AUTO\_INCREMENT数据列生成出一组负序列值。但AUTO\_INCREMENT数据列的这种用法并没有得到支持和提倡，其结果也没有保证。根据我个人的经验，负序列值在MySQL不同版本中的行为是有差异的，即使在MySQL的某个版本里达到了目的，一旦升级到了一个新的版本，情况有可能会发生变化。（换句话说，用AUTO\_INCREMENT数据列来处理正整数序列以外的任何东西都有可能产生无法预料的后果。请大家千万注意！）

- 千万不要自以为是地认为把AUTO\_INCREMENT添加到一个数据列声明上就能得到无穷无尽的序列编号。这种想法是错误的。AUTO\_INCREMENT序列要受制于具体的数据列类型的取值范围。比如说，如果使用的是一个TINYINT数据列，那么序列编号的最大值是127，一旦达到了这个上限，就会因“键字重复”错误而使操作失败。如果使用的是TINYINT UNSIGNED数据列，那么序列编号的上限值是255。
- 清除一个数据表的全部内容将使有关序列从1开始重新编号，即使所使用的数据表类型通常不允许再次使用AUTO\_INCREMENT编号值也会如此。这种情况会在使用下面这两条语句时发生：

```
DELETE FROM tbl_name;
TRUNCATE TABLE tbl_name;
```

序列之所以重新从1开始编号，是因为MySQL会把全表删除操作（也叫“数据表完全删除操作”）优化为“先把数据表里的数据行和索引一次性地全部清除掉，然后再重新创建这个数据表”的动作，而不是一个一个地去删除数据表里的数据行，这将导致序列编号信息的丢失。如果想删除所有的数据行却又想保留序列编号信息，就必须像下面这样用带一条WHERE子句的DELETE语句来抑制这种优化：

```
DELETE FROM tbl_name WHERE 1;
```

这将迫使MySQL为每个数据行都进行一次条件表达式的求值操作，这样可以逐个删除数据行。

### 2.3.7 强制MySQL不要复用已经用过的序列值

MySQL的某些数据表类型允许再次使用从序列顶部被删除过的编号值。那么，怎样才能让这些数据表里的序列编号也严格地保持递增而不会被再次使用呢？有一种解决方案是这样的：另外创建一个专门用来生成AUTO\_INCREMENT值的数据表，并做到永远不去删除这个数据表里的任何记录——让这个数据表根本就不存在编号值被再次使用的问题。当需要在主数据表里生成一条新记录时，先在那个专门用来生成序列编号的数据表里插入一个NULL值以生成一个严格递增的编号值，然后，在往主数据表里插入新记录的时候，利用LAST\_INSERT\_ID()函数取得这个编号值并把它赋值给主数据表里用来存放序列编号的那个数据列，如下所示：

```
INSERT INTO ai_tbl SET ai_col = NULL;
INSERT INTO main_tbl SET id=LAST_INSERT_ID() ... ;
```



### 2.3.8 给数据表增加一个序列编号数据列

假设已经创建了一个数据表并在其中存入了一些信息，如下所示：

```
mysql> CREATE TABLE t (c CHAR(10));
mysql> INSERT INTO t VALUES('a'),('b'),('c');
mysql> SELECT * FROM t;
```

c
a
b
c

现在，想给这个数据表增加一个序列编号数据列。于是，用一条ALTER TABLE语句给它增加了一个AUTO\_INCREMENT数据列，而它的声明定义应该与用CREATE TABLE语句创建一个这样的数据列时所使用的一样，如下所示：

```
mysql> ALTER TABLE t ADD i INT AUTO_INCREMENT NOT NULL PRIMARY KEY;
mysql> SELECT * FROM t;
```

c	i
a	1
b	2
c	3

请注意，MySQL自动地完成了AUTO\_INCREMENT数据列里的序列编号赋值工作，根本用不着你亲自去做这件事。

### 2.3.9 重新编排现有的序列编号

如果数据表里已经有了一个AUTO\_INCREMENT数据列，但现在想对它重新进行编号以消除因删除数据记录而在序列中产生的断裂带。最简单的办法是：先丢弃（删除）该数据列，然后再重新把它添加出来。MySQL会在重新添加该数据列的时候自动完成序列编号的赋值工作，就像前面那个例子里一样。

先来创建一个数据表t，其中有一个AUTO\_INCREMENT数据列i：

```
mysql> CREATE TABLE t (c CHAR(10), i INT NOT NULL AUTO_INCREMENT PRIMARY KEY);
mysql> INSERT INTO t (c)
-> VALUES('a'),('b'),('c'),('d'),('e'),('f'),('g'),('h'),('i'),('j'),('k');
mysql> DELETE FROM t WHERE c IN('a','d','f','g','j');
mysql> SELECT * FROM t;
```

+-----+-----+	
c	i
+-----+-----+	
b	2
c	3
e	5
h	8
i	9
k	11
+-----+-----+	

下面的ALTER TABLE语句将完成先丢弃、再重新创建数据列i的工作：

```
mysql> ALTER TABLE t
      -> DROP i,
      -> ADD i INT UNSIGNED AUTO_INCREMENT NOT NULL,
      -> AUTO_INCREMENT = 1;
mysql> SELECT * FROM t;
```

+-----+-----+	
c	i
+-----+-----+	
b	1
c	2
e	3
h	4
i	5
k	6
+-----+-----+	

子句AUTO\_INCREMENT = 1将使序列从1开始重新进行编号。如果是MyISAM数据表（或者是MySQL 4.1及以后的HEAP数据表），可以用一个不等于1的数字使序列从另外一个值开始重新进行编号。如果是其他的数据表类型，就不必写出AUTO\_INCREMENT = n子句了，这是因为它们不允许像这样来设定序列编号的初始值，序列将从1开始重新进行编号。

不过，虽然重新编排现有序列编号的工作很容易完成，但往往没有必要去这么做。要知道，MySQL并不在意编号序列里有没有断裂带，而你也不会因为重新编排了序列编号而在性能方面得到任何好处。

### 2.3.10 在不使用AUTO\_INCREMENT机制的情况下生成序列编号

另一种生成序列编号的办法是根本就不使用AUTO\_INCREMENT数据列。这种办法需要用到带参数的LAST\_INSERT\_ID()函数。（带参数的LAST\_INSERT\_ID()函数最早出现于MySQL 3.22.9版本。）如果用函数LAST\_INSERT\_ID(expr)去插入或者修改了一个数据列，紧接着又调用了一次不带参数的LAST\_INSERT\_ID()函数，第二次函数调用所返回的就将是表达式expr的值。换句话说，MySQL将把表达式expr的值视为一个新生成的AUTO\_INCREMENT值，这就使得可以先生成一个序列编号，然后再在后面的会话进程中对它进行检索，而无需顾虑这个编号值是

否会受到其他客户（程序）的活动的影响。

这种策略的一种用法是创建一个只有一个数据行的数据表，其中的值将在每次需要一个序列编号的时候被更改一次。比如说，可以像下面这样来创建和初始化这个数据表：

```
CREATE TABLE seq_table (seq INT UNSIGNED NOT NULL);
INSERT INTO seq_table VALUES(0);
```

上面这些语句创建了一个只有一个数据行的数据表seq\_table，并把seq的初始值设置为0。这个数据表的使用方法是这样的：先生成一个新的序列编号，再把它检索出来，如下所示：

```
UPDATE seq_table SET seq = LAST_INSERT_ID(seq+1);
SELECT LAST_INSERT_ID();
```

上面这条UPDATE语句将检索出seq数据列的当前值，给它加上一个1以生成序列中的下一个编号。利用函数LAST\_INSERT\_ID(seq+1)来生成新编号值的做法将使它被MySQL视为一个新生成的AUTO\_INCREMENT值，而这个新编号值又可以用不带参数的LAST\_INSERT\_ID()函数检索出来。LAST\_INSERT\_ID()函数与具体的客户（程序）是相关联的，即使其他客户（程序）在发出UPDATE语句和SELECT语句的时间间隔里又生成了其他的序列编号，所检索到的也仍将是刚生成的那个新编号值。

利用这一策略，还能生成间隔步长不等于1（甚至是负步长）的序列编号来。比如说，反复执行下面这条语句，就能生成间隔步长等于100的序列编号来：

```
UPDATE seq_table SET seq = LAST_INSERT_ID(seq+100);
```

而反复执行下面这条语句将生成按1递减（间隔步长等于-1）的序列编号来：

```
UPDATE seq_table SET seq = LAST_INSERT_ID(seq-1);
```

利用这一技巧，只要给seq数据列设置一个适当的初始值，就能生成一个从任意值开始编号的序列来。

上面的讨论描述了如何利用一个单数据行的数据表来建立一个序列计数器。这个办法也确实不错，可如果需要用到多个计数器，还是为每个计数器分别创建一个单数据行的数据表就会毫无必要地弄出太多的数据表了。比如说，很多网站都在自己的Web网页上放置一个访问计数器，用来统计和显示“本网页已被访问n次”的信息，此时大可不必为每个网页去分别创建一个计数器数据表。

既然不想创建多个计数器数据表，就只能是设法把多个计数器安排在同一个数据表里了。可以这样做：创建一个有两个数据列的数据表，其中一个数据列用来存放计数值，另一个数据列用来保存各计数器的名字，这些名字必须是独一无二的。另外，还要求助于LAST\_INSERT\_ID()函数，但现在需要用各计数器的名字把它们区分开来。这个数据表应该是下面这样的：

```
CREATE TABLE counter
(
    name VARCHAR(255) BINARY NOT NULL,
    PRIMARY KEY (name),
    value INT UNSIGNED
);
```

把用来存放计数器名字的名称数据列声明为VARCHAR(255) BINARY类型，这对给计数器起任何名字来说都应该是足够的了。同时，为了避免计数器的名字出现重复，把它声明为一个PRIMARY KEY，这就要求那些会用到这个数据表的“东西”给计数器起的名字不得重复。就这个Web计数器的例子来说，上一句话里的“东西”指的就是那些网页，而要想给那些网页的计数器分别起一个不重复的名字，它们的文档路径名应该是一个不错的选择。为了区分网页文档路径名里的大小写字母，我们还给name数据列加上了BINARY属性（如果所使用的系统不区分文件名里的字母大小写情况，就可以去掉这个属性）。

在创建出counter数据表之后，还得往里面插入一些数据行，每个数据行对应着一个将要用到的网页计数器。比如说，如果想设置一个对应于网站主页的访问计数器，可以像下面这样做：

```
INSERT INTO counter (name,value) VALUES('index.html',0);
```

这条语句创建了一个名为'index.html'（因为这只是一个示例，所以这里没有写出index.html文件完整的路径名）的计数器，并把它的计数值初始化为0。以后，当需要为这个网页生成一个新的序列编号时，只需根据它的路径名查出与之对应的计数值，用LAST\_INSERT\_ID(expr)函数递增之，再用LAST\_INSERT\_ID()函数检索出那个新编号值就可以了，如下所示：

```
UPDATE counter SET value = LAST_INSERT_ID(value+1) WHERE name = 'index.html';
SELECT LAST_INSERT_ID();
```

另一种做法是直接对计数器值做加法而不使用LAST\_INSERT\_ID()函数：

```
UPDATE counter SET value = value+1 WHERE name = 'index.html';
SELECT value FROM counter WHERE name = 'index.html';
```

但这种做法有个小问题：在发出UPDATE命令之后、发出SELECT命令之前，如果有其他的客户（程序）对计数器进行了操作，就将得到一个错误的结果。要解决这个问题也不难——只需使用一个事务（transaction）或者在这两条语句的前后加上LOCK TABLES和UNLOCK TABLES命令就行了，这样，在使用计数器的这段时间内，其他客户（程序）将被阻塞。但很明显，用LAST\_INSERT\_ID()函数来完成这项任务要简便得多。LAST\_INSERT\_ID()函数的返回值是与具体的客户（程序）相关联的，所以所得到的计数器值肯定就是当初插入的那个，不可能来自别的客户（程序）。同时，因为不必使用事务，所以代码也容易写一些；又因为不必锁定数据表，所以其他客户（程序）也不会被阻塞。

## 2.4 MySQL对字符集的支持

那些由各种字符构成的数据值的含义必须用一个给定的字符集来做出解释，字符集决定了哪些字符可以用来表示数据。字符集里的字符都有一个固定的排位次序，这个排位次序又会对很多与字符值有关的操作产生影响，比如说：

- 比较操作：<、<=、=、>、>=、>
- 排序操作：ORDER BY、MIN()、MAX()
- 分组操作：GROUP BY、DISTINCT

字符集对服务器的某些方面（比如哪些字符可以用在数据库、数据表和数据列的名字里）也



有影响, 这些名字大都要由服务器默认字符集里的字母数字类字符构成(见第3.1节中的讨论)。

不同的MySQL版本对字符集的支持程度是不一样的。在MySQL 4.1之前的版本里, 服务器在任意时刻只能使用一种字符集。在MySQL 4.1及以后的版本里, 不仅服务器能够同时使用多种字符集, 而且在服务器、数据库、数据表、数据列以及字符串常数等多个级别上也能对字符集进行设定了。比如说, 如果想让数据表里的数据列都默认地使用latin1字符集, 但有一个数据列要使用希伯来语, 还有一个数据列要使用希腊语, 可以这样做。还可以查出当前有多少种字符集可供使用, 或者把数据从一种字符集转换为另一种字符集。

本节的学习重点是如何使用MySQL服务器所支持的字符集。如果因为需要使用某个字符集而需要对服务器进行配置, 请参阅第11章中的有关内容。为了让大家的旧数据表能享受到MySQL 4.1版本中的新功能, 本章还将介绍一些与字符集有关的MySQL 4.1升级方案。

#### 2.4.1 MySQL 4.1之前版本对字符集的支持

在MySQL 4.1之前, MySQL里的数据值没有明确的字符集。字符串常数和数据列里的取值都是用服务器的默认字符集来解释的。默认情况下, 就是在对MySQL的服务器程序进行编译时选定的那个字符集(通常是latin1), 但这个内建值可以在启动MySQL服务器的时候通过--default-character-set选项加以改变。这是一种非常简单的解决方案, 但有着极大的局限性, 例如, 可能无法用同一个数据表里不同的数据列来分别保存使用了不同字符集的数据。

这种单字符集模型还会导致与索引有关的问题。比如说, 如果在已经完成了数据表的创建和字符数据的加载工作之后又改变了服务器的默认字符集, 就可能会遇到一些麻烦。这是因为: 索引值其实是一些排好了序的数据, 而字符数据列的排序顺序由服务器在建立索引时使用的默认字符集来决定。有些字符在不同的字符集里有着不同的排序顺序。如果服务器在往数据表里加载数据时使用的是一种字符集, 在检索那些数据时使用的却是另一种字符集, 那么, 根据前一种字符集而确定的索引值排序就很可能在后一种字符集里被认为是不正确的。更糟糕的是, 当再往数据表里添加新数据行时, MySQL服务器将按字符在后一种字符集里的排序顺序对根据前一种字符集排序得到的索引值进行修改。这样, 那些需要用到索引的查询就很可能工作异常。

要想彻底解决这一问题, 就必须用后一种字符集的排序顺序去重建每一个现有数据表的每一个基于字符的索引。这种重建工作可以用好几种办法来完成:

- 先用mysqldump程序导出数据表里的数据, 再清除数据表里的内容, 最后用导出文件重新填满它。数据表的索引将在重新加载数据时得到重建。这个办法适用于各种数据表类型。
- 丢弃那些索引, 然后再重建它们。可以用ALTER TABLE命令或者用DROP INDEX和CREATE INDEX命令来做这件事。这个办法也适用于各种数据表类型, 但要想把有关索引精确地重建出来, 必须知道有关它们的确切定义才行。
- MyISAM数据表的索引可以用myisamchk程序的--recover和--quick选项再加上一个用来设定新字符集的---set-character-set选项进行重建。还可以用mysqlcheck程序的--repair和---quick选项或者一条带QUICK选项的REPLACE TABLE语句来重建那些索引。mysqlcheck程序和REPLACE TABLE命令用起来更方便一些, 因为索引重建工作是由服务器完成的, 而服务器知道自己现在正使用着哪一种字符集。myisamchk程序则必须离线脱

机运行，所以必须明确地设定将要使用哪一种字符集。

在服务器的默认字符集被改变之后，不论打算采用哪种办法来重建索引，都必须由自己进行具体的操作，而正是这一点让我们感到不愉快。MySQL 4.1彻底解除了这一负担。

## 2.4.2 MySQL 4.1及以后版本对字符集的支持

MySQL 4.1及其后续版本对字符集的支持有了很大的改观并提供了很多新的功能：

- 支持服务器同时使用多种字符集。
- 允许在服务器、数据库、数据表、数据列和字符串常数等多个级别上对字符集进行设定，不再像以前那样仅限于服务器级别了：
  - 如果需要改变数据库级的字符集设置，可以使用ALTER DATABASE命令。
  - 如果需要改变数据表和数据列级的字符集设置，可以使用CREATE TABLE和ALTER TABLE子句。
- 利用MySQL 4.1提供的函数和操作符，既可以把数据值从一种字符集转换为另一种字符集，也可以查出某项数据使用的是哪一种字符集。
- 新增加的COLLATE操作符使我们能够按某一种字符集的排序顺序来处理另一种字符集里的数据。
- 新增加的SHOW CHARACTER SET命令能够把服务器当前支持的字符集全都列出来。
- 当服务器切换使用另一种字符集时，会自动对索引进行重新排序。
- 通过utf8和ucs2字符集提供了Unicode支持。
- 新增了很多字符集。

MySQL现在还不支持：1）在同一个字符串里混用来自不同字符集的字符；2）在同一数据列里混用不同的字符集。但只要愿意，就完全可以利用Unicode字符集（它能用来表示很多种人类语言）来实现自己的多语言支持。

### 1. 字符集的设定

可以在多个级别上（从服务器的默认字符集到每一个字符串）对字符集进行设定：

- 服务器的默认字符集是在编译期间选定的，但这个内建值可以在启动MySQL服务器的时候通过--default-character-set选项加以改变。
- 如果想为某个数据库设定一个默认字符集，请使用下面的语句：

```
ALTER DATABASE db_name DEFAULT CHARACTER SET charset;
```

其中，*charset*或者是服务器所支持的某个字符集的名字，或者是DEFAULT。DEFAULT表示该数据库将使用MySQL服务器的默认字符集作为其数据库级字符集。

- 如果想为某个数据表设定一个默认字符集，需要在创建该数据表时加上一个CHARACTER SET子句，如下所示：

```
CREATE TABLE tbl_name (...) CHARACTER SET = charset;
```

其中，*charset*或者是服务器所支持的某个字符集的名字，或者是DEFAULT。DEFAULT表示该数据表将使用它所在的数据库的默认字符集作为其数据表级字符集。

- 数据表里的数据列可以通过其CHARACTER SET属性明确地设定一个字符集。如下所示：

```
c CHAR(10) CHARACTER SET charset
```

其中, *charset*必须是服务器所支持的某个字符集的名字,而不能是DEFAULT。如果没有写出CHARACTER SET属性,就将使用数据表级字符集。允许设定字符集的数据列类型是CHAR和VARCHAR(不能带BINARY属性)及TEXT类型。

- 字符串常数可以通过下面的表示法转换到一个给定的字符集,其中的*charset*必须是服务器所支持的某个字符集的名字:

```
_charset str
```

下面两种表示法将生成两个分别使用latin1\_de和utf8字符集的字符串:

```
_latin1_de 'abc'
```

```
_utf8 'def'
```

这种表示法只适用于那些括在引号中的字符串。十六进制常数、字符串表达式或者数据列的取值都不允许使用这种表示法。不过,可以用CONVERT()函数把任何形式的字符串转换到一个给定的字符集:

```
SELECT CONVERT(str USING charset);
```

如果想对一种字符集的数据按另一种字符集的排序顺序进行排序,就需要使用COLLATE操作符。比如说,假设c是一个使用latin1字符集的数据列,但又想按latin1\_de字符集的排序顺序对它进行排序,那就得使用下面这样的命令:

```
SELECT c FROM t ORDER BY c COLLATE latin1_de;
```

## 2. 确定哪些字符集可用

MySQL 4.1及后续版本对字符集的支持包括一些用来获得各级别信息的语句:

- 在服务器级别,可以用下面这条语句查知都有哪些查询字符集可供使用:

```
SHOW CHARACTER SET;
```

如果想知道服务器的默认字符集是哪一个,请使用下面这个查询:

```
SHOW VARIABLES LIKE 'character_set';
```

- 对于给定数据库,它的数据库级字符集可以用下面这个命令查出来:

```
SHOW CREATE DATABASE db_name;
```

如果这条语句的输出里没有表明一个字符集,那么这个数据库的字符集或者尚未设定,或者被明确地设定为DEFAULT。

- 对于给定数据表,它的字符集用下面两个命令都能查出来:

```
SHOW CREATE TABLE tbl_name;
```

```
SHOW TABLE STATUS LIKE 'tbl_name';
```

- 数据列上的字符集设定情况可以用下面这几条语句之一查出来:

```
DESCRIBE tbl_name;
```

```
SHOW COLUMNS FROM tbl_name;
```

```
SHOW CREATE TABLE tbl_name;
```

- 如果想知道与某特定字符串、字符串表达式或者数据列值相关联的字符集是哪一个，可以使用CHARSET()函数：

```
SELECT CHARSET(str);
```

### 3. Unicode支持

语言不同，人们为之开发的编码方案也就不同。这既是有这么多字符集的原因，也是很多与字符有关的问题的根源。比如说，有的字母会出现在好几种语言里，但不同的编码方案却可能用不同的编码值来代表它。再比如说，不同的语言往往需要使用不同个数的字节来表示字符，Latin-1字符集里的字符只需一个字节来表示，而汉语或日语里的“字符”则需要多个字节来表示。

Unicode的目标是把各种字符编码方案融合在一起，以便各种语言都能以一种整齐一致的方式被表示出来。MySQL中的Unicode支持是通过两个字符集提供的：

- UTF-8：一种可变长度的编码格式，需要用1~4个字节来表示一个字符。（UTF是“UCS Transformation Format”的缩写，中文意思是“UCS变形格式”。而这个UCS本身又是“Universal Character Set”的缩写，中文意思是“通用字符集”。）MySQL使用的utf8字符集没有收录那些需要用4个字节来表示的字符，但今后的版本可能会把它们添加进来。
- UCS2：是MySQL支持的另一个Unicode字符集。ucs2字符集中的每个字符要用两个字节来表示，高位字节在前。这个字符集没有收录需要用两个以上字节来表示的字符。

### 4. 把老数据表转换为MySQL 4.1格式

把服务器升级到MySQL 4.1或更高版本以后，那些老数据表虽然还能继续使用，却很可能享受不到新增字符集支持所带来的全部好处。因此，最好把它们全都转换为4.1格式，具体操作步骤可以参见第11章。

## 2.5 选择数据列类型

前面的第2.2节对MySQL中的数据列类型以及各个类型的基本特点做了比较详细的介绍，比如它们能够容纳的数据类型、它们的存储空间占用量，等等。当打算创建一个数据表的时候，面对这么多的数据列类型，怎样才能做出最正确的选择呢？本节将对一些比较重要的考虑因素进行讨论，希望能够对大家有所帮助。

字符串类型算是最基本的数据列类型了。数值和日期可以被表示为字符串，可以把任何数据放到字符串里。那么，为什么不把数据列全都声明为字符串类型呢？我们来看一个简单的例子。假如有一些看起来像数字的数据，可以把它们表示为字符串，可应不应该这样做呢？如果这样做了，会发生什么呢？

首先，用字符串来表示数字值往往需要占用更多的存储空间，用一个数值数据列来存储数字值效率更高。其次，MySQL对数字和字符串的处理方式是不同的，所以它们在查询结果中也会有差异。比如说，数字和字符串的排序顺序就不一样：数值2小于数值11，但字符串'2'却大于字符串'11'。如果想把一个字符串数据列用在数值上下文里，就必须像下面这样迂回一下：

```
SELECT col_name + 0 as num ... ORDER BY num;
```



给col\_num列加上一个0就能强制进行数值排序，可有必要绕这么一个圈子吗？在某些情况下，这是一个很有用的技巧，可要是每次进行数值排序的时候都不得不这样做，就让人觉得麻烦了。强迫MySQL把一个字符串数据列当做数值数据列还有一些隐含的副作用。首先，数据列里的每个值都必须做一次“字符串-数值”的转换，而这是一种很低效的操作。其次，让数据列参加运算将迫使MySQL不能使用建立在该数据列上的索引，而这又将使查询变得更慢。要是从一开始就把这些数据放到一个数值数据列里，这两种会让性能降低的情况就都不会发生了。在挑选数据列类型的时候，不仅要考虑它们都能用来表示哪些数据，还必须考虑到它们的存储空间占用量、查询的处理方式、处理性能等多方面因素。

刚才的例子揭示出了选择数据列类型时的几个问题。在为数据列挑选一种类型的时候，下面这些因素有助于迅速找到最佳选择：

- **这个数据列将用来存放哪一种数据？**数值？字符串？还是日期？这个问题不难回答，但必须要问。完全可以把任意类型的值都表示为字符串。但正如我们刚才看到的那样，如果能用一种更适当的类型来保存数字值（日期和时间值也是如此），就会得到更好的性能。给数据（尤其是别人的数据）挑选一种适当的类型并不像想像得那么简单。在动手创建数据表之前，一定要把哪些数据将被放到哪些数据列里弄清楚，如果是在替别人创建数据表的话，这个问题就更重要了。要知道，了解的信息越多，做出的决策就越正确。
- **数据值是否都位于某个区间范围内？**如果它们是整数，那它们中间会不会有负数？如果没有，就可以使用UNSIGNED。如果它们是字符串，那它们的各种取值是否能构成一个有限集合？如果是，就应该考虑使用ENUM或SET类型。  
给定一个数据列类型，取值范围越大，存储空间占用量也就越大。那么，多“大”的类型才能满足具体要求呢？对于数字值，其类型的大小要根据它们值的大小来选定；对于字符串，其类型的大小则要根据它们的长短来选定。比如说，如果字符串数据全都短于10个字符，就没有选用一个CHAR(255)数据列的必要。
- **有没有性能和效率方面的问题？**有些类型的处理效率要比其他类型的高。一般来说，数值操作的效率要比字符串操作的效率高。短字符串的比较操作要比长字符串完成得更快，有关的磁盘操作也更少。在ISAM和MyISAM数据表里，固定长度的数据列类型要比可变长度的数据列类型的性能更高。
- **打算如何对有关数据进行比较？**字符串的比较操作分为区分字母大小写和不区分字母大小写两种情况。所做的决定还将影响到字符串的排序和分组操作，因为它们都是以比较操作为基础的。
- **是否要在某个数据列上建立索引？**如果有这个打算，就要在挑选数据表类型和数据列类型的时候留出余地——因为不同的数据表类型在索引的建立和使用方面有着不同的做法。比如说，ISAM数据表不允许在BLOB和TEXT数据列上建立索引，而允许建立索引的数据列又必须被定义为NOT NULL（也就是说，如果打算在ISAM数据表的某个数据列上建立索引，就不能在其中使用NULL值）。

下面将依次对这几个问题进行详细分析。但在开始之前，先做几点说明。首先，在创建数据表的时候，当然是想用最适当的数据列类型把它创建得尽善尽美；如果当时的选择被以后的

事实证明不是最佳的，那也不是什么世界末日，完全可以通过ALTER TABLE命令把它改成更好的类型。这种修改可能很简单，比如说，如果发现某个整数数据列里的值超出了原来的取值范围，那只需把SMALLINT改成MEDIUMINT就行了。这种修改也可能很复杂，比如把某个CHAR数据列整个地改为一个枚举类型的ENUM数据列。在MySQL 3.23及以后的版本里，可以用PROCEDURE ANALYSE()函数对数据表里的数据列进行分析。对于给定的数据列，这个命令除了能查看到它的最大值、最小值等统计信息外，还会给出一个它认为最优的数据列类型来：

```
SELECT * FROM tbl_name PROCEDURE ANALYSE();
```

这个查询的输出有助于发现是否能把某些数据列声明为一个较“小”的类型，这既能提高有关查询的执行效率，同时也能节约不少的存储空间。

### 2.5.1 这个数据列将用来存放哪一种数据

在挑选数据列类型的时候，首先要考虑并回答好这个问题。一般说来，这个问题的答案似乎很直观：把数值保存在数值数据列里，把字符串保存在字符串数据列里，把日期和时间保存在日期和时间数据列里。如果是整数，就使用一个整数类型；如果有小数部分，就使用一个浮点类型。但事情并不总是这么简单。只有洞察到有关数据的本质，才能明智地选出最佳的类型。比如说，如果将被放入数据表的数据都是自己的，那它们的用途当然是自己最清楚了；可如果是在替别人创建数据表，最好多花点时间去向别人请教这些事情。建议能多问就多问，一定要把每一个数据列将用来存放哪一种数据的问题搞明白。

我们来看一个例子。假设所创建的数据表里有一个用来记录“降水量”的数据列。那么，它是一个数值呢，还是一个在大部分时间里是数字（即在一般情况下是数字，但偶尔会是其他类型）的值呢？比如说，在电视台的天气预报节目里，我们经常会听到“降水量”这个词。它在多数时候是一个数字（如“降水量0.25英寸”），但播音员偶尔也会说“降水量少”。播音员可以在天气预报节目里这样说，但在数据库里怎样来表示它呢？一种办法是把这个“少”量化为一个数值，这样就能用一个数值数据列类型来记录降水量；另一种办法是使用一个字符串数据列类型，这样就能直接用“少”这个字来表示这种情况。还可以使用一些更复杂的机制，比如同时使用一个数值数据列和一个字符串数据列来记录降水量的情况，但这种办法既会增加数据表的理解难度，也会增加数据表的查询难度——只要还有其他选择，我想是没有多少人会采用这一方案的。

如果让我来解决这一问题的话，我大概会把所有的数据都保存为数值，然后根据一定的条件把它们转换为必要的显示格式。比如说，如果我们把小于0.01英寸（但不等于0）的降水量归入“降水量少”的范畴内，就可以用下面这条语句来显示有关的数据：

```
SELECT IF(precip>0 AND precip<.01,'trace',precip) FROM ... ;
```

在某些场合，虽然已经能够确定有关数据都是一些数值，但仍需进一步判断是要使用一个整数类型还是要使用一个浮点类型。应该把有关数据的计量单位和它们的精确度考虑进来。整数单位够不够用？是否需要使用小数？这类问题能帮助在整数和浮点数据列类型之间做出选择。比如说，在表示重量的场合，如果只需精确到磅，那么一个整数数据列就已经足够了。如果需

要精确到盎司，那就得考虑使用浮点数据列了。在某些场合，使用多个数据列也是一种值得考虑的策略——比如说，可以用一个数据列来记录磅值，用另一个数据列来记录盎司值。

高度也是一种有多种表示方案的数值：

- 字符串方案，比如用'6-2'表示“6英尺2英寸”。这个方案的好处是有关数据容易阅读和理解（当然比“74英寸”好），但要想对这些数据进行求和或求平均值之类的算术运算就不容易了。
- 两个数值数据列方案，一个记录英尺值，另一个记录英寸值。在算术运算方面有了改善，但两个数据列怎么也不如一个数据列用起来简便。
- 一个数值数据列方案，以英寸为单位进行记录。这是最适合数据库使用的方案，同时也是最不适合人类阅读和理解的方案。但大家也不要忘了，数据的表示格式不一定非得是使用它们时需要的格式。可以利用MySQL提供的各种函数把数据转换成想要的格式。综合以上分析，这应该是表示高度值的最佳方案。

另一种比较常见的数值类信息是财务数据。财务计算必须精确到分，所以财务数据似乎应该被表示为浮点数，但因为FLOAT和DOUBLE类型存在着四舍五入的问题，所以它们只适用于对精确度要求不高的场合。可是，人们对于金钱往往是锱铢必较的，所以必须知道哪种数据列类型能够提供完美的精确度。选择不外乎以下两种：

- 可以把财务数据定义为一个DECIMAL(M, 2)类型，其中的M必须足以表示有关数据中的最大金额，这将使得能够以精确到小数点后面两位的方式表示一个浮点值。

使用DECIMAL类型的好处是：因为有关数据被表示为字符串的形式，所以就不存在四舍五入的问题。它的缺点是：字符串操作要比数值操作的效率低。

- 可以把财务数据以分为单位保存到一个整数类型的数据列里去。这个办法的好处是：整数运算的速度非常快。它的缺点是：在输入或输出中，需要对有关数值进行乘以100或除以100的转换。

我们再来看看日期和时间信息的情况。在对日期信息进行处理的时候，首先要把其中是否还包括时间的问题搞清楚。必须弄清这样一个问题：日期值里到底包不包括时间？在MySQL里，不存在时间值可有可无的数据类型——它的DATE类型不允许包括时间，而DATETIME类型又不能省略时间。如果日期数据里的时间值的确是可有可无的，就得使用一个DATE数据列来记录日期，再另外使用一个TIME数据列来记录时间，然后用TIME数据列里的NULL值来表示“时间值缺失”的情况。如下所示：

```
CREATE TABLE mytbl
(
    date DATE NOT NULL,      # date is required
    time TIME NULL          # time is optional (may be NULL)
);
```

之所以要在这里强调日期值是否包括时间的问题，是因为它们在某些应用项目里有着举足轻重的作用。比如说，如果需要通过两个数据表里的日期数据列把它们关联起来进行查询，就必须把这个问题弄清楚。



我们来看一个例子。假设正在做一些试验并准备把试验结果保存到数据库里。需要先做一个初步试验，然后再根据初步试验的结果有选择地做一些后续试验。如果把初步试验的结果保存在一个主数据表里，把后续试验的结果保存在一个从数据表里，这样，在检索操作中，将需要把这两个数据表通过匹配的试验编号和试验日期关联起来。

于是就产生了这样一个问题：是只需要使用日期，还是需要同时使用日期和时间？这个问题的答案取决于是否会在同一天里进行一次以上的试验（初步试验+后续试验）。如果不会，那就只需记下试验当天的日期值；如果会，那还得把试验当天里的一个时间值（比如试验的开始时间）也记下来，或者使用一个DATETIME数据列，或者使用一个DATE数据列加上一个TIME数据列（这两个数据列都不能为空）。假如有可能在同一天里进行一次以上的试验却没有记下一个时间值，就无法在检索操作中把从数据表里的后续试验结果与主数据表里的初步试验结果正确地关联在一起。

也许有些人会这样说：“我不需要记录一个时间值，因为我决不会在同一天里进行一次以上的试验。”如果他们真的能说到做到，那所有的担心就是多余的。可又有谁能保证这种情况不会发生呢？如果没有做好准备工作，等真的出现在同一天里进行了一次以上试验的情况时，再想办法就太晚了。

这种困境可以用给主、从数据表各增加一个TIME数据列的办法来补救，但这往往又会陷入另一种困境：如果没有保留有关试验的原始结果，就很难对数据表里的现有记录做出调整。没有试验结果的原始记录，就无法区分同一天里的多次试验。即使保留着试验结果的原始记录，如果项目已经进行了相当长的时间，数据表里已经有了很多条记录，想不出差错地把它们都调整好也不是件容易的事。总而言之，在动手创建数据表之前先把有关数据的特点弄明白是最稳妥的策略——如果是在为别人创建数据表，千万要把这一点向对方解释清楚。

有时候，数据本身可能就不完整，这也将影响到对数据列类型的挑选。比如说，正为钻研家谱学而收集整理着有关人等的生卒日期，所收集到的这些生卒日期可能只有年份或者只有年份和月份——总之，不是一个完整的日期值。如果使用的是一个DATE数据列，就无法把这些不完整的日期数据插入到数据表里。如果想让数据表能把所收集到的生卒日期信息（无论它们完整与否）都照单全收，就应该用不同的数据列来分别保存年、月、日——把生卒日期中知道的部分填写进去，把不知道的部分留为NULL值。在MySQL 3.23及以后的版本里，又多了一种选择：DATE值里的日子或者月份和日子允许为0，而这种“模糊的”日期正好可以用来表示不完整的日期值。

### 2.5.2 数据值是否都位于某个区间范围内

为有关数据选定了数据列的基本类型（整数、浮点数、字符串、日期）之后，还需要进一步根据它们的取值范围来选定一种具体的数据列类型。以整数值为例，它们的取值范围决定了应该具体选用哪一种整数类的数据列类型。如果整数值全都位于0~1000的区间内，考虑范围将从SMALLINT一直延伸到BIGINT。如果这个区间的上限扩大到了200万，就不能再考虑SMALLINT了，考虑范围只能是从MEDIUMINT到BIGINT。剩下的事是从这些候选方案里挑选出一个最终的数据列类型来。



当然，完全可以简单地用最“大”的数据列类型（例如，整数类数据列类型中的BIGINT）来存放数据。但从原则上讲，应该从那些能够满足要求的数据列类型里挑选一个最“小”的来使用。这样做的好处有二：一是能把数据表的存储空间占用量压缩到最小，二是能获得更好的性能——因为尺寸较小的数据列通常要比尺寸较大的数据列更容易处理，其处理速度也更快。（一般来说，较小的值只需较少的磁盘活动就能读取完毕，索引缓存区里容纳的键值也会更多，这都有助于使利用索引的查询能够以更快的速度得到完成。）

如果无法预料有关数据的取值范围，那么，为了应付可能出现的最坏情况，就只能依靠猜测或者简单地选用BIGINT等最“大”的数据列类型了。（如果依靠猜测而选定的数据列类型被事实证明是过小了的话，还有办法来补救——用ALTER TABLE命令来“加大”数据列。）

在第1章里，为考试记分项目创建了一个名为score的数据表，其中有一个用来记录考试和测验分数的score数据列。为了简化第1章中的讨论，当时是把它声明为INT类型的。但根据上面的说法，如果考试分数全都位于0~100这个区间内的话，TINYINT UNSIGNED无疑是一种更节省存储空间因而也是更好的选择。

数据的取值范围还会影响到为它们选定的数据列类型的某些属性。比如说，如果数据不会出现负值，就可以加上一个UNSIGNED属性；否则，就不能这样做。

字符串类型不像数值数据列那样有大小意义上的取值范围，它们只有长短之分。字符串数据的最大长度决定了应该具体选用哪一种字符串类的数据列类型。如果字符串数据全都短于256个字符，则可以使用CHAR、VARCHAR、TINYTEXT或TINYBLOB。如果想使用更长的字符串，就只能在TEXT或BLOB类型里挑一个，CHAR和VARCHAR将不再可选。

如果字符串数据只有有限的可取值，那还可以考虑选用ENUM或SET数据列类型。在这种情况下，它们往往是更好的选择，因为它们在MySQL的内部是被表示为数字的。ENUM或SET数据列上的操作是以数值方式进行的，这使它们比其他字符串类型有着更高的处理效率。同时，因为它们比其他字符串类型更紧凑，所以还能节省大量的存储空间。

在分析有关数据的取值范围的时候，如果能用“总是”和“从不”来描述它们（比如“总是小于1000”或“从不出现负值”），那可是再好不过的了，因为它们能把数据列类型的候选范围限制得更窄。但这两个词在实际生活中经常会被滥用，大家要特别注意这一点。如果是在为别人创建数据表，那么，当向对方询问有关数据的取值范围并听到他们说出这两个词时，请一定要多加小心。相当一部分人嘴里的“总是”和“从不”都不是这里讨论的意义，当他们说他们的数据总是如何如何的时候，这个词的真正含义往往是“几乎总是”。

比如说，假设为一所学校设计一个数据表，校方这样说：“我们的考试分数总是在0~100的区间内。”根据这句话，选择了TINYINT UNSIGNED类型来保存全都是非负值的考试分数。可到最后却发现校方在录入考试分数的时候会用-1来表示“学生因病缺勤”的情况。校方当初并没有明确地说用NULL值来代表考试缺勤的情况也是可以接受的。现在，必须让-1也能被记录到数据表里，可这却是UNSIGNED数据列所不允许的。（幸好还能用ALTER TABLE命令进行补救！）

有时候，只要再多问一个简单的问题就能帮你迅速做出选择：你们说的这种情况有例外吗？如果有例外——哪怕只出现一次，也必须事先做好准备。在与别人探讨数据表的设计问题时，经常会遇到对方认为“因为例外情况很少发生，所以它们无关紧要，说不说都无所谓”的

情况。但因为数据表要由你来负责创建，所以你不能像他们那样模棱两可。你关心的问题不是“例外情况会发生得多么频繁”而是“例外情况会不会发生”，只要例外情况有可能发生，就必须提前准备好应对措施。

### 2.5.3 有没有性能和效率方面的问题

所选定的数据列类型会对今后的查询操作产生多方面的影响。但如果参照下列指导意见把问题考虑得很周全，所选定的数据列类型就能帮助MySQL更有效地处理数据表。

#### 1. 数值操作/字符串操作

数值操作一般都比字符串操作快。拿比较操作来说，数值之间的比较只需一个操作，字符串之间的比较则要一个字节一个字节或者一个字符一个字符地进行，字符串越长，比较次数也就越多。

如果某个字符串数据列的可取值能够构成一个有限集合，就可以考虑使用ENUM或SET类型以利用数值操作的好处。这两种类型在MySQL内部是以数值形式来表示的，所以有较高的处理效率。

多考虑几种不同的字符串表示方法。如果能把字符串表示为数值，就能大幅改善其处理性能。比如说，像192.168.0.4这样的4字段IP地址通常是用字符串来表示的，但完全可以用一个4字节的INT UNSIGNED类型来存放它，每个字节存放一个字段，这既可以节约存储空间，又能加快检索的速度。可是，被表示为INT值的IP数字不适用于模式匹配操作，诸如判断某IP数字是否位于某给定子网之内的工作将很难有效地完成。因此，在替有关数据选择表示形式的时候，不仅要考虑怎样才能节约存储空间，还必须考虑它们今后的具体用途。（就数字化IP地址的例子而言，无论选用的是哪一种表示形式，都可以利用INET\_ATON()和INET\_NTOA()函数对它们进行相互转换。）

#### 2. 小类型/大类型

小类型的处理速度比大类型的更快。这是因为小类型占用的存储空间比较少，因而磁盘读写方面的开销也就比较小。这在字符串上体现得尤其明显——字符串数据的处理时间与它们的长度有着直接的关系。

对于固定长度的字符串类型，应该选用能满足有关数据取值范围要求的最小类型。比如说，如果MEDIUMINT能满足有关数据的取值范围要求，就不要使用BIGINT类型；如果FLOAT能满足有关数据的精确度要求，就不要使用DOUBLE类型。对于可变长度的字符串类型，也有节约存储空间的办法。比如说，一个BLOB要占用2个字节来保存数据值的长度，而一个LONGBLOB则要占用4个字节。如果有关数据的长度从不超过64KB，选用BLOB将在每个数据身上节约2个字节。（TEXT类型也是如此。）

#### 3. 固定长度类型/可变长度类型

固定长度类型和可变长度类型对性能的影响是不同的，但它们的具体影响还要取决于所选用的数据表类型。

对于MyISAM和ISAM数据表，固定长度类型通常要比可变长度类型的处理速度更快。这是因为：

- 如果数据表里有可变长度的数据列，在经过一些删除或修改操作后，因为数据记录的长度各不相同，就将产生更多的碎片。要想保持性能不会降低，就必须定期执行OPTIMIZE TABLE命令。固定长度的数据行则不存在这一问题。
- 如果数据表里都是固定长度的数据列，在发生损坏的时候就更容易得到修复。这种数据表里的每条记录都有着同样的长度，所以很容易确定各条记录的起始点——它必定等于记录长度的某个整数倍数，这是可变长度的数据行所不具备的特点。这是一个与查询处理无关的性能问题，但肯定能加快数据表的修复速度。

如果MyISAM或ISAM数据表里有可变长度的数据列，把它们转换为固定长度的数据列将有助于改善其性能，因为固定长度的数据记录要更容易处理。但在进行这类转换之前，必须考虑以下问题：

- 虽然固定长度的数据列处理起来比较快，但它们占用的存储空间却比较大。CHAR(*n*)数据列里的每个值都要占用*n*个字节（空数据也是如此），不足*n*个字节的数据在被放入数据表之前要用空格来补足。VARCHAR(*n*)数据列占用的存储空间比较少，因为其中的每个值将根据“按需分配”的原则获得存储空间，每个值还要再加上一个用来记录其长度的字节。如此看来，选用CHAR数据列还是选用VARCHAR数据列的关键其实就是想节约时间还是节约空间的问题。如果看重速度，就应该选用CHAR数据列以利用固定长度数据列的高性能；如果看重空间，就应该选用VARCHAR数据列。作为一项基本原则，可以假设固定长度的数据行即使会消耗较多空间，也足以在性能方面得到补偿。但在某些极其关键的应用项目里，可以考虑同时采用这两种办法来创建数据表，然后再通过一些测试来最终决定它们中的哪一种更有利于实际情况。
- 不能只转换一个可变长度数据列，必须全部转换它们。准确地说，必须用一条ALTER TABLE命令同时把它们转换为固定长度的数据列，要不然就没有效果。
- 有时候，即使想使用固定长度的数据列，也无法做到这一点。比如说，如果字符串数据的长度超过了255个字节，就不存在与之相对应的固定长度类型了。

在InnoDB数据表里，固定长度和可变长度的数据行都是以同样方式存储的（一个数据行表头加上一段用来实际存放有关数据的存储空间，表头里是指向该数据行各个列值的指针）。这意味着固定长度的数据行并没有性能方面的优势，决定性能优劣的主要因素将是各数据行所占用的存储空间总量。换句话说，InnoDB数据表里的可变长度数据行往往有着更快的处理速度，因为它们占用的存储空间比较少，要求的磁盘I/O活动也比较少。

#### 4. 可索引类型

索引能加快查询速度，所以应该尽量选用那些允许进行索引的类型——至少要为那些打算用在检索条件表达式里的数据列做这样的选择。具体做法见第2.5.5节。

#### 5. NULL类型/NOT NULL类型

给定一个数据列，如果把它定义为NOT NULL，那么，MySQL就不必在检索过程中去检查该数据列里的数据是不是一个NULL值，这将加快处理速度。同时，这还能在每个数据行上节约一个位的存储空间。避免在数据列里使用NULL值还有助于简化查询语句（因为不再有数据取值为NULL的特殊情况需要去考虑了），而简单的查询语句又往往能更快地得到处理。

### 2.5.4 打算如何对有关数据进行比较

用来保存字符串数据的数据列类型往往会对有关的比较和排序操作是否区分大小写情况产生影响，这主要由有关数据列里包含的是二进制字符串（区分大小写）还是非二进制字符串（即普通的字符串，不区分大小写）来决定。表2-15列出了MySQL中的二进制字符串类型和相应的普通字符串类型。有些类型（BLOB和TEXT）的“二进制性”隐含在该类型的名称里，其他类型（CHAR和VARCHAR）是否具有“二进制性”则要视有关数据列的声明定义里是否有关键字BINARY而定。

表2-15 MySQL中的二进制字符串类型和非二进制字符串类型

二进制类型	非二进制类型
CHAR(M) BINARY	CHAR(M)
VARCHAR(M) BINARY	VARCHAR(M)
TINYBLOB	TINYTEXT
BLOB	TEXT
MEDIUMBLOB	MEDIUMTEXT
LOB	LONGTEXT

如果想让某个数据列既能用在区分大小写的比较操作里，也能用在不区分大小写的比较操作里，可以使用一个非二进制类型。然后，在区分大小写的比较操作里，用关键字BINARY强制MySQL把字符串当做二进制字符串来看待。假设mycol是一个CHAR数据列，我们先来看一个不需要区分大小写情况的字符串比较操作：

```
mycol = 'ABC'
```

再来看两个需要区分大小写情况的字符串比较操作（注意：BINARY操作符具体施加在哪个字符串上并不重要）：

```
BINARY mycol = 'ABC'
mycol = BINARY 'ABC'
```

如果想按某种非字母表顺序对字符串数据进行排序，可以考虑使用一个ENUM数据列。ENUM值的排序顺序是在声明有关数据列时给出的合法取值列表中的次序。因为ENUM值的次序是完全由你来决定的，所以完全可以让有关的字符串数据按想要的任何顺序进行排序。

### 2.5.5 是否要在某个数据列上建立索引

索引可以加快MySQL的查询处理速度。如何选择和建立索引将在第4章详细讨论，但作为一项基本原则，那些会出现在查询命令WHERE子句里的数据列通常是最值得考虑的索引候选目标。

如果想在某个数据列上建立索引或者想把它用在某个多数据列索引里，那么在为这个数据列挑选数据列类型的时候，就必须遵守一些规定。比如说，有些数据表类型（InnoDB和ISAM）不允许在BLOB或TEXT数据列上建立索引。又比如说，在MySQL 3.23.2之前，索引只能建立在被声明为NOT NULL的数据列上。如果遇到这类限制，可以采用以下几种迂回手段：

- 如果想在BLOB或TEXT数据列上建立索引而数据表类型却不允许这样做，请检查数据是



否有长度超过255个字节的。如果没有，先把这个BLOB或TEXT数据列替换为一个VARCHAR数据列，然后再建立索引。可以用VARCHAR(255) BINARY来替换BLOB，用VARCHAR(255)来替换TEXT。

- 绕过NOT NULL限制的办法是另外安排一个特殊值来表示当初用NULL来表示的含义。比如说，对于一个DATE数据列，可以用'0000-00-00'来表示当初用NULL来表示的“无日期”含义；对于一个字符串数据列，如果其中的正常数据都是非空字符串，就可以用空字符串来表示当初用NULL来表示的“没有数据”含义；对于一个数值数据列，如果其中的正常数据都是非负值，就可以用-1来表示当初用NULL来表示的含义（当然，要是想这么做，那就不能把这个数据列声明为UNSIGNED了）。

### 2.5.6 数据列类型选择问题的内在联系

在挑选数据列类型的时候，不能孤立地看待各个因素，它们之间有着内在的联系。比如说，数值类型的取值范围与它们的存储空间占用量就有着密切的关系：随着取值范围的扩大，存储空间占用量也会相应地增大，而这又会对性能产生影响。基于这种考虑，这里再集中探讨一下在声明和使用AUTO\_INCREMENT数据列时应该注意的几个问题。当需要一组独一无二的序列编号的时候，我们就会想到AUTO\_INCREMENT数据列。这是一个谁都会想到的解决方案，但由此而产生的对数据列类型、索引以及NULL值等因素的影响就不一定尽人皆知了。

- 从本质上讲，AUTO\_INCREMENT是一种只应该用在整数类型上的数据列属性，这立刻就把选择范围限制在从TINYINT到BIGINT之间的整数类数据列类型里了。
- 从本质上讲，AUTO\_INCREMENT数据列只应该用来生成正整数序列，所以应该把它们声明为UNSIGNED。
- AUTO\_INCREMENT数据列必须有索引。进一步讲，为了避免序列编号出现重复，AUTO\_INCREMENT数据列上的索引还必须是惟一的。这就意味着必须把它声明为一个PRIMARY KEY或者一个UNIQUE索引。
- AUTO\_INCREMENT数据列必须具备NOT NULL属性。

综上所述，将不能像下面这样去声明一个AUTO\_INCREMENT数据列：

```
mycol arbitrary_type AUTO_INCREMENT
```

而是应该像下面这样声明之：

```
mycol integer_type UNSIGNED AUTO_INCREMENT NOT NULL,  
PRIMARY KEY (mycol)
```

或者像下面这样声明之：

```
mycol integer_type UNSIGNED AUTO_INCREMENT NOT NULL,  
UNIQUE (mycol)
```

## 2.6 表达式求值与类型转换

MySQL允许使用以各种常数、函数调用、数据列名称为“零件”，以各种操作符（例如，算

术操作符和比较操作符)为“粘合剂”而构成的表达式,同一表达式里的“零部件”还允许用括号进行归组。表达式可以用在很多地方,其中又以查询结果的输出明细表和SELECT语句的WHERE子句里最为多见。请看下面这个例子,它与第1章里统计美国总统逝世年龄时使用的查询命令差不多:

```
SELECT
    CONCAT(last_name, ', ', first_name),
    (YEAR(death) - YEAR(birth)) - IF(RIGHT(death,5) < RIGHT(birth,5),1,0)
FROM president
WHERE
    birth > '1900-1-1' AND DEATH IS NOT NULL;
```

在这条查询命令的SELECT部分和WHERE子句里,使用了多个表达式。在DELETE和UPDATE语句的WHERE子句以及INSERT语句的VALUES()子句里,也经常能看到大量的表达式。

在遇到表达式的时候,MySQL将对它进行求值并产生一个求值结果,比如说,表达式 $(4 * 3) / (4 - 2)$ 的求值结果是6。表达式的求值过程往往伴随着各种类型转换操作,比如说,在一个需要DATE值的上下文里,MySQL将自动地把数值960821转换为日期值'1996-08-21'。

在本节里,将着重介绍MySQL表达式的各种用法和MySQL在对表达式进行求值的过程中所使用的各种类型转换规则。这里将讨论MySQL的每一种操作符,但论及的函数却只是一小部分——因为MySQL的函数实在是太多了。有关MySQL函数的详细介绍请参阅附录C。

### 2.6.1 书写表达式

表达式可以简单到只是一个常数,例如:

0	一个数值常数
'abc'	一个字符串常数

表达式里允许出现函数调用。有些函数带有输入参数(即括号里的值),有些则不带。如果某个函数的输入参数多于一个,就要用逗号把它们分隔开。在调用函数的时候,输入参数之间允许出现空格,但函数名与紧随其后的左括号之间则不允许出现空格<sup>①</sup>。如下所示:

NOW()	一个不带输入参数的函数
STRCMP('abc', 'def')	一个带两个输入参数的函数
STRCMP('abc', 'def')	输入参数两边出现空格,合法
STRCMP ('abc', 'def')	函数名后面出现空格,非法

如果函数名后面出现空格,MySQL就会把它解释为一个数据列名。(在MySQL里,函数名不是保留字。如果你愿意,就可以把它们用做数据列名。)但把函数名用做数据列名的情况不多见,所以其结果通常是一条提示“语法错误”的出错信息。

表达式里还允许出现数据列名和数据列值。在最简单的场合,即MySQL能够根据上下文清

① 不过,如果在启动MySQL服务器的时候使用了--ansi或--sql-mode = IGNORE\_SPACE选项,就会使MySQL把函数名全都视为保留字并允许函数名的后面出现空格了。

楚地知道某个数据列是属于哪一个数据表的时候，可以只给出该数据列的名字。在下面两条SELECT语句里，因为它们都只用到了一个数据表，所以尽管两条语句里的数据列名字完全一样。MySQL也分得清它们到底属于哪一个数据表：

```
SELECT last_name, first_name FROM president;
SELECT last_name, first_name FROM member;
```

如果MySQL无法根据数据列的名字把它们区分开来，就必须在数据列名字的前面加上它所在的数据表的名字作为修饰。如果MySQL无法确定数据表属于哪一个数据库，就必须在数据表名字的前面加上它所在的数据库的名字作为修饰。即使在不会导致歧义的上下文中，也可以采用这种更为明确的形式来书写表达式。如下所示：

```
SELECT
    president.last_name, president.first_name,
    member.last_name, member.first_name
FROM president, member
WHERE president.last_name = member.last_name;
SELECT sampdb.student.name FROM sampdb.student;
```

最后，MySQL允许你利用以上介绍的这些“零件”（常数、函数调用、数据列名）构造出各种复杂的表达式。

### 1. 操作符的种类

为了把各种表达式“零件”粘合在一起，MySQL提供了几种类型的操作符。算术操作符包括常见的加法操作符（+）、减法操作符（-）、乘法操作符（\*）、除法操作符（/）和一个不怎么常见的求余操作符（%），见表2-16。两个操作数都是整数的加法、减法、乘法将使用BIGINT（64位）整数值来进行，计算结果将用于整数上下文的除法和求余操作也将使用BIGINT（64位）整数值来进行；除此之外的其他操作都将使用DOUBLE浮点值来进行。但要注意的是，如果整数运算的结果超出了BIGINT（64位）整数值的表示范围，其结果将无法预料。（事实上，应该尽量避免超出63位整数值的表示范围，因为符号也要占用一个位。）

表2-16 算术操作符

操 作 符	语 法	含 义
+	$a + b$	加法，两个操作数之和
-	$a - b$	减法，两个操作数之差
-	$-a$	单操作数求负操作符，把操作数取为负值
*	$a * b$	乘法，两个操作数之积
/	$a / b$	除法，两个操作数之商
%	$a \% b$	求余，除法操作的余数

逻辑操作符（见表2-17）用来对逻辑表达式的真（非0）、假（0）进行求值。如果操作数的值无法确定（比如“1 AND NULL”的情况），逻辑表达式还可能被求值为NULL。MySQL允许使用C语言风格的“&&”、“||”、“!”操作符作为其AND、OR、NOT操作符的替代形式。请大

家特别要注意“||”操作符，ANSI SQL把“||”规定为字符串合并操作符，但MySQL却把它用做逻辑或操作符<sup>①</sup>。

表2-17 逻辑操作符

操 作 符	语 法	含 义
AND 或 &&	a AND b a && b	逻辑与；若两个操作数同时为真，则结果为真
OR 或	a OR b a    b	逻辑或；只要有一个操作数为真，则结果为真
XOR	a XOR b	逻辑异或；若有且仅有一个操作数为真，则结果为真
NOT 或 !	NOT a ! a	逻辑非；若操作数为假，则结果为真

不熟悉MySQL的人在看到下面这个表达式的求值结果是0而不是一个字符串时难免会大吃一惊：

'abc' || 'def' → 0

这个操作会先把字符串'abc'和'def'转换为整数，而这两个字符串的转换结果都将是0。在MySQL里，如果想把这两个字符串合并在一起，就必须用CONCAT('abc','def')函数来进行：

CONCAT('abc','def') → 'abcdef'

位操作符（见表2-18）中的“&”、“|”、“^”分别用来完成对操作数进行按位与、按位或、按位异或操作（逻辑异或操作符“XOR”和按位异或操作符“^”是从MySQL 4.0.2版本才开始引入的）；而“<<”和“>>”则用来完成位的左移和右移操作。位操作都要使用BIGINT（64位）整数值来完成。

表2-18 位操作符

操 作 符	语 法	含 义
&	a & b	按位与；若两操作数的同一位同时为1，则结果中的该位为1
	a   b	按位或；若两操作数的同一位有一个为1，则结果中的该位为1
^	a ^ b	按位异或；若两操作数的同一位分别为1和0，则结果中的该位为1
<<	a << b	把a中的各个位左移b个位置
>>	a >> b	把a中的各个位右移b个位置

比较操作符（见表2-19）包括用来判断数值和字符串的大小或排序顺序的操作符、用来进行模式匹配的操作符以及测试NULL值的操作符。“<=>”操作符是MySQL独有的，它最早出现于MySQL 3.23版本。

① 如果想让“||”操作符的行为符合ANSI标准，请在启动MySQL服务器的时候使用--ansi或--sql-mode = PIPES\_AS\_CONCAT选项。



表2-19 比较操作符

操 作 符	语 法	含 义
=	a = b	若两个操作数相等, 则结果为真
<=>	a <=> b	若两个操作数相等, 则结果为真 (可用于NULL值)
!= 或 <>	a != b 或 a <> b	若两个操作数不等, 则结果为真
<	a < b	若a小于b, 则结果为真
<=	a <= b	若a小于或等于b, 则结果为真
>=	a >= b	若a大于或等于b, 则结果为真
>	a > b	若a大于b, 则结果为真
IN	a IN (b1, b2, ...)	若a等于b1、b2……中的某一个, 则结果为真
BETWEEN	a BETWEEN b AND c	若a在b值和c值之间 (包括b和c), 则结果为真
NOT BETWEEN	a NOT BETWEEN b AND c	若a不在b值和c值之间 (包括b和c), 则结果为真
LIKE	a LIKE b	SQL模式匹配; 若a匹配b, 则结果为真
NOT LIKE	a NOT LIKE b	SQL模式匹配; 若a不匹配b, 则结果为真
REGEXP	a REGEXP b	正则表达式匹配; 若a匹配b, 则结果为真
NOT REGEXP	a NOT REGEXP b	正则表达式匹配; 若a不匹配b, 则结果为真
IS NULL	a IS NULL	如果a是NULL值, 则结果为真
IS NOT NULL	a IS NOT NULL	如果a不是NULL值, 则结果为真

最早出现于MySQL 3.23版本的BINARY操作符可以用来把一个字符串转换为一个二进制字符串, 其效果通常是使该字符串上的比较或排序操作开始区分字母的大小写情况。在下面的例子里, 第一个比较操作是不区分字母大小写情况的, 第二和第三个比较操作则区分字母的大小写情况:

```
'abc' = 'Abc'                                → 1
BINARY 'abc' = 'Abc'                          → 0
'abc' = BINARY 'Abc'                          → 0
```

不存在与NOT BINARY操作符对应的操作符。如果想让某个数据列既能用在区分字母大小写情况的上下文里也能用在不区分字母大小写情况的上下文里, 可以先把它声明为一个不区分大小写情况的数据列类型, 然后在需要区分字母大小写情况的比较操作中用BINARY操作符来修饰它。另一个办法是: 先把这个数据列声明为一个区分大小写情况的数据列类型, 然后在不需区分字母大小写情况的比较操作中用UPPER()或LOWER()函数把两个操作数都转换为同样的大小写。如下所示:

```
UPPER(col_name) < UPPER('Smith')
LOWER(col_name) < LOWER('Smith')
```

在那些不需要区分字母大小写情况的比较操作里, 根据所使用的字符集, 有可能出现两个 (甚至更多个) 不同的字符被认为是彼此“相等”的情况。比如说, 字母“E”和“É”就可能会被比较和排序操作认为是“相等”的。再看二进制字符串的比较操作, 它是以依次比较两个操作数相同位置上的字符的数值编码的方式进行的, 所以能够区分出字母的大小写情况。

模式匹配是一种“模糊的”比较操作, 它不需要给出一个精确无误的参照值就能把有关数据检索出来。MySQL提供了两种模式匹配机制: 一种叫SQL模式匹配, 利用LIKE操作符以及

通配符“%”（能与任意个字符序列相匹配）和“\_”（只能与一个字符相匹配）进行匹配；另一种叫REGEXP模式匹配，利用REGEXP操作符和正则表达式（regular expression）进行匹配，该机制中的正则表达式与grep、sed、vi等UNIX程序所使用的正则表达式非常相似。模式匹配只能使用各种模式匹配操作符才能完成；等号操作符（=）只能用来判断两个数据是否相等，不具备模式匹配能力。与模式匹配概念相对立的“模式不匹配”操作可以用NOT LIKE或NOT REGEXP操作符来完成。

除不同的操作符和不同的模式字符（也就是通配符）外，两种模式匹配机制还在以下方面有着重要的差异：

- LIKE操作符一般不区分字母的大小写情况，除非它的操作数里至少有一个是二进制字符串。REGEXP操作符也是如此，但在MySQL 3.23.4以前的版本里，REGEXP操作符却一直是区分字母的大小写情况的。
- 只有整个字符串都匹配时，SQL模式才算匹配成功；而只要能在字符串里找到匹配，正则表达式模式就算匹配成功。

与LIKE操作符配合使用的模式匹配里可以包括通配符“%”和“\_”。比如说，模式'Frank%'将匹配任何一个以'Frank'开头的字符串：

```
'Franklin' LIKE 'Frank%'          → 1
'Frankfurter' LIKE 'Frank%'       → 1
```

通配符“%”能与任意长度的字符序列（包括空字符序列）匹配，所以模式'Frank%'完全能够与'Frank'匹配：

```
'Frank' LIKE 'Frank%'            → 1
```

这意味着模式'%能够与任何一个字符串（包括空字符串）匹配，但“%”不能匹配NULL值。事实上，任何一个模式都不能与NULL值匹配：

```
'Frank' LIKE NULL                → NULL
NULL LIKE '%'                   → NULL
```

MySQL的LIKE操作符通常是不区分字母大小写情况的，除非它的操作数里至少有一个二进制字符串。也就是说，在默认情况下，模式'Frank%'既能与'Frankly'匹配，也能与'frankly'匹配；但在二进制字符串的比较操作中，它就只能与'Frankly'匹配了，如下所示：

```
'Frankly' LIKE 'Frank%'          → 1
'frankly' LIKE 'Frank%'          → 1
BINARY 'Frankly' LIKE 'Frank%'   → 1
BINARY 'frankly' LIKE 'Frank%'   → 0
```

这种行为与ANSI SQL的LIKE操作符是不一样的，ANSI SQL的LIKE操作符是区分字母大小写情况的。

通配符可以出现在模式里的任意位置。比如说，模式'%bert'只能与'Englebert'、'Bert'、'Albert'等以'bert'结尾的字符串相匹配；模式'%bert%'除了能与上述字符串相匹配外，还能与'Berthold'、'Bertram'、'Alberta'等以'bert'开头或者其中包含有'bert'的字符串相匹配；模式'b%t'则能与'Bert'、'bent'、'burnt'等任何一个以字母'b'开头以字母't'结尾的字符串相匹配。

LIKE操作符的另一个通配符是下划线字符“\_”，它只能用来匹配单个字符。比如说，模式‘\_\_\_’将与任何一个有且仅有三个字符的字符串相匹配；而模式‘c\_t’将与‘cat’、‘cot’、‘cut’，甚至‘c\_t’（因为‘\_’能够匹配自身）相匹配。

如果想对字符“%”或“\_”进行匹配，就必须给它们加上一个前导的反斜线字符（即写成“\%”或“\\_”的样子）以取消其特殊含义，如下所示：

```
'abc' LIKE 'a%c'           → 1
'abc' LIKE 'a%c'           → 0
'a%c' LIKE 'a%c'           → 1
'abc' LIKE 'a_c'           → 1
'abc' LIKE 'a\_c'          → 0
'a_c' LIKE 'a\_c'          → 1
```

MySQL还可以使用正则表达式进行匹配。这种模式匹配使用的操作符是REGEXP而不是LIKE。下面是一些比较常用的正则表达式模式字符：

- 句点字符“.”，用来匹配任何单个的字符：

```
'abc' REGEXP 'a.c'         → 1
```

- 方括号“[...]”构造，用来匹配在方括号内部出现的任何字符：

```
'e' REGEXP '[aeiou]'       → 1
'f' REGEXP '[aeiou]'       → 0
```

- 连字符“-”，用来给出一个字符范围——在字符“-”的两端分别写出这个范围的起始字符和结束字符即可。如果在这个字符范围的前面再加上一个“^”字符，就可以用它来匹配那些不属于这一范围的字符了，如下所示：

```
'abc' REGEXP '[a-z]'       → 1
'abc' REGEXP '[^a-z]'       → 0
```

- 星号字符“\*”，用来匹配“前一字符任意次数的连续重复出现”。比如说，模式‘x\*’将匹配连续出现的任意个数的‘x’字符：

```
'abcdef' REGEXP 'a.*f'     → 1
'abc' REGEXP '[0-9]*abc'    → 1
'abc' REGEXP '[0-9][0-9]*'  → 0
```

这里所说的“任意次数”也包括0次在内（即允许字符根本没有出现），而这正是上面第二个表达式被求值为1的原因。

- ‘^pat’和‘pat\$’形式的模式分别用来匹配子串‘pat’出现在字符串的开头或末尾的情况；而‘^pat\$’形式的模式可以用来匹配子串‘pat’完全匹配整个字符串的情况，如下所示：

```
'abc' REGEXP 'b'           → 1
'abc' REGEXP '^b'          → 0
'abc' REGEXP 'b$'          → 0
'abc' REGEXP '^abc$'       → 1
'abcd' REGEXP '^abc$'      → 0
```

MySQL允许使用数据列的取值作为REGEXP模式，不过，如果该数据列有几种取值的话，

这种做法要比使用一个常数形式的模式来进行匹配的速度慢——数据列的取值每变化一次，MySQL都不得不重新检查一次有关模式并把它再次转换为内部格式。

MySQL的正则表达式模式匹配还有其他一些特殊的模式字符，关于这方面的详细信息可以在附录C里查到。

## 2. 操作符的优先级

在对表达式进行求值的时候，MySQL会根据操作符的优先级来决定表达式各“零件”的求值次序。优先级较高的操作符会先于其他操作符得到求值。比如说，乘法和除法的优先级就要比加法和减法的高。请看下面两个例子，它们的求值结果之所以相同，就是因为操作符“\*”和“/”将在操作符“+”和“-”之前得到求值：

```
1 + 2 * 3 - 4 / 5          → 6.2
1 + 6 - .8                 → 6.2
```

我们把操作符按优先级从高到低的顺序依次列在下面这份清单里。在这份清单里，同一行上的操作符都有着相同的优先级，优先级较高的操作符将在优先级较低的操作符之前得到求值，而优先级相同的操作符将按从左至右的顺序依次得到求值。

优先级	操 作 符
高	BINARY、COLLATE
	NOT、!
	^
	XOR
	-（一元求负操作符）、~（一元位求反操作符）
	*, /、%
	+, -
	<<、>>
	&
	<, <=, =, <=>, !=, <>, >=, >, IN, IS, LIKE, REGEXP, RLIKE
	BETWEEN、CASE、WHEN、THEN、ELSE
	AND、&&
	OR、
	:=
低	

括号可以用来改写操作符的优先级并改变表达式各“零件”的求值次序，如下所示：

```
1 + 2 * 3 - 4 / 5          → 6.2
(1 + 2) * (3 - 4) / 5      → -0.6
```

## 3. 表达式中的NULL值

要特别注意表达式里出现的NULL值，因为其结果往往会出乎预料。下面是一些基本的注意事项。

如果把NULL值用做算术操作符或位操作符的操作数，其求值结果将是NULL：

```
1 + NULL          → NULL
1 | NULL          → NULL
```



如果把NULL值用做逻辑操作符的操作数，那么，除非真的有一个确切的结果，其求值结果也将是NULL<sup>①</sup>。

1 AND NULL	→ NULL
1 OR NULL	→ 1
0 AND NULL	→ 0
0 OR NULL	→ NULL

如果把NULL值用做比较操作符的操作数，那么，除专门用来处理NULL值的“<=>”、“IS NULL”和“IS NOT NULL”操作符以外，其他比较操作符的求值结果都将是NULL：

1 = NULL	→ NULL
NULL = NULL	→ NULL
1 <=> NULL	→ 0
NULL LIKE '%'	→ NULL
NULL REGEXP '.*'	→ NULL
NULL <=> NULL	→ 1
1 IS NULL	→ 0
NULL IS NULL	→ 1

如果把NULL值用做函数的输入参数，那么，除支持NULL值作为其输入参数的那些函数以外，其他函数通常都会返回一个NULL值。比如说，IFNULL()就是一个支持NULL值作为其输入参数的函数，它将根据输入参数的具体情况返回一个真值或假值。再比如说，STRCMP()函数不支持NULL值作为其输入参数，如果它发现传递给它的输入参数是NULL值，就将返回一个NULL而不再是一个真值或假值。

在排序操作中，NULL值都将被集中到一起，但它们是出现在非NULL值的前面还是出现在后面则取决于具体的MySQL版本，这一点已经在第1章的“对查询结果进行排序”小节里讨论过了。

## 2.6.2 类型转换

在表达式里，如果某个数据值的类型与上下文所要求的类型不相符，MySQL就会根据将要进行的操作自动地对该数据值进行类型转换。下面是几种比较常见的需要进行类型转换的场合：

- 根据某操作符的求值规则把其他类型的操作数转换为“正确的”类型。
- 根据某函数对输入参数的预期把其他类型的输入参数转换为“正确的”类型。
- 根据某数据列的类型定义把其他类型的数据值转换为“正确的”类型并把它插入到该数据列里去。

还可以利用类型转换操作符或函数来明确地进行类型转换。

下面就是一个需要在求值之前进行类型转换的表达式，它由一个加法操作符(+)和两个操作数(1和'2')构成：

```
1 + '2'
```

<sup>①</sup> 在MySQL 3.23.9以前的版本里，逻辑操作符的NULL值操作数被当做一个假值来使用，这种行为现在被认为是一个程序漏洞。

既然两个操作数的类型不一致（一个是数值，另一个是字符串），MySQL就得对其中的某一个进行类型转换才能使它们一致起来。那么，应该对哪一个操作数进行类型转换呢？在这个例子里，“+”是一个数值操作符，它要求两个操作数都是数值类型，所以MySQL将先把字符串'2'转换为数值2，然后再求出这个表达式的最终结果3。我们再来看一个例子。CONCAT()函数能够把多个字符串合并为一个更长的字符串，不管输入参数是什么类型，这个函数都将把它们当做字符串来对待。于是，当把一些数值传递给CONCAT()函数的时候，会先把它们转换为字符串，然后再把它们合并起来作为自己的返回值，如下所示：

```
CONCAT(1,2,3)                                → '123'
```

如果CONCAT()函数又是一个更大的表达式的“零件”，就需要经过更多的类型转换才能得到最终的结果。请看下面这个表达式：

```
REPEAT('X',CONCAT(1,2,3)/10)                 → 'XXXXXXXXXXXXX'
```

这个表达式的求值过程是这样的：首先，CONCAT(1, 2, 3)返回字符串'123'；接着，'123'/10又被转换为123/10（因为“/”是一个算术操作符）。表达式123/10在浮点上下文里的结果是12.3，但因为REPEAT()函数需要一个整数值来给出字符'X'的重复次数，所以123/10在这里进行的是整数除法，计算结果等于12。最后，REPEAT('X', 12)把字符'X'重复了12次，也就是上面所看到的字符串结果。

在对表达式进行求值的时候，MySQL会尽可能地进行类型转换而不是报告错误，这一原则请大家一定要记住。根据上下文，MySQL会把不符合求值要求的数据类型（数值、字符串、日期和时间）尽可能地转换为所需要的数据类型，但数据并不是总能从一种类型转换为另一种类型的。如果某给定类型的数据不能转换为目标类型里的合法数据，MySQL就会用一些“默认值”来充当转换结果。比如说，数值上下文里的字符串'abc'根本就不能转换为一个合法的数值，所以MySQL会把这次类型转换的结果取为“默认值”0。同样地，如果某项数据看上去根本不像是日期和时间值，则在日期和时间上下文里的类型转换结果就将是目标日期和时间类型的“零”值。比如说，把字符串'abc'转换为一个DATE值的结果将是DATE类型的“零”值——'0000-00-00'。反之，因为任何数据都可以用字符串来表示，所以把其他类型的数据值转换为字符串通常不会有什么麻烦。

除上面介绍的这些从一种数据类型到另一种数据类型的“大”转换外，MySQL还能够同种数据类型中的子类型之间做一些“小”转换。比如说，如果在整数上下文里使用了一个浮点数，MySQL就会按四舍五入的规则把它转换为整数。反过来也是如此，整数可以毫无问题地被当做浮点数来使用。

如果上下文没有明确地表明需要的是一个数值，十六进制常数就会被当做字符串来对待。在字符串上下文里，每两个十六进制数字会被转换为一个字符，最终结果将是一个字符串。我们来看一些例子：

```
0x61                                → 'a'
0x61 + 0                            → 97
X'61'                               → 'a'
X'61' + 0                           → 97
```

CONCAT(0x61)	→ 'a'
CONCAT(0x61 + 0)	→ '97'
CONCAT(X'61')	→ 'a'
CONCAT(X'61' + 0)	→ '97'

十六进制常数在比较操作中的类型取决于具体使用的MySQL版本。在MySQL 3.23.22及以后的版本里，比较操作中的十六进制常数被视为数值，如下所示：

0x0a = '\n'	→ 0
0xaaab < 0xab	→ 0
0xaaab > 0xab	→ 1
0x0a = 10	→ 1

在MySQL 3.23.22之前的版本里，如果不是与数值进行比较，那么比较操作中的十六进制常数被视为二进制字符串。因此，上面几个比较操作在老版本的MySQL中的执行结果有所不同，如下所示：

0x0a = '\n'	→ 1
0xaaab < 0xab	→ 1
0xaaab > 0xab	→ 0
0x0a = 10	→ 1

有些操作符会把自己的操作数强制性地转换为该操作符预期的类型，而不管那些操作数原来是哪一种类型。算术操作符就是这样做的，它们要求自己的操作数必须是数值，并会强制进行相应的转换：

3 + 4	→ 7
'3' + 4	→ 7
'3' + '4'	→ 7

在把字符串转换为数字的时候，字符串里包含有一个数值并不够。MySQL不会检查整个字符串以找出其中包含的数值，它只检查字符串的开头。如果字符串的开头部分不是数值，转换结果就将是0，如下所示：

'1973-2-4' + 0	→ 1973
'12:14:01' + 0	→ 12
'23-skidoo' + 0	→ 23
'-23-skidoo' + 0	→ -23
'carbon-14' + 0	→ 0

需要提醒大家特别注意的是，MySQL的“字符串 - 数字”类型转换规则从3.23版本开始发生了重大改变。目前，看起来像数值的字符串都会被转换为浮点数值；在3.23版本之前，它们却是按四舍五入的规则被转换为整数的。如下所示：

'-428.9' + 0	→ -428.9 (MySQL ≥ 3.23)
'-428.9' + 0	→ -429 (MySQL < 3.23)

逻辑操作符和位操作符对操作数的要求比算术操作符还要严格。它们不仅要求操作数是数值，还要求它们必须是整数，并会按照这一要求进行相应的类型转换。这意味着像0.3这样的浮点数（虽然是一个非零值）将不再被视为是逻辑真值，因为把0.3转换为整数所得到的结果将是

0。在下面几个表达式里，小于1的操作数都没有被视为逻辑真值：

0.3 OR .04	→ 0
1.3 OR .04	→ 1
0.3 AND .04	→ 0
1.3 AND .04	→ 0
1.3 AND 1.04	→ 1

这样的转换也发生在IF()函数里，它要求自己的第一个参数是整数，那些会被四舍五入为0的浮点数将被视为逻辑假值，如下所示：

IF(1.3, 'non-zero', 'zero')	→ 'non-zero'
IF(0.3, 'non-zero', 'zero')	→ 'zero'
IF(-0.3, 'non-zero', 'zero')	→ 'zero'
IF(-1.3, 'non-zero', 'zero')	→ 'non-zero'

因此，如果需要对浮点数做出正确的比较，最好是使用一个明确的比较操作：

IF(0.3>0, 'non-zero', 'zero')	→ 'non-zero'
-------------------------------	--------------

模式匹配操作通常在字符串上进行，这也意味着完全可以把MySQL的模式匹配操作符用在数值上，因为数值能在模式匹配过程中被转换为字符串，如下所示：

12345 LIKE '1%'	→ 1
12345 REGEXP '1.*5'	→ 1

数量意义上的比较操作符(<、<=、=等等)是上下文敏感的，即它们将根据操作数的具体类型来进行求值。在下面这个表达式里，两个操作数都是数值，所以它们之间的比较是数值型的：

2 < 11	→ 1
--------	-----

在下面这个表达式里，两个操作数都是字符串，所以它们之间的比较是字符串型的：

'2' < '11'	→ 0
------------	-----

但在下面两个表达式里，因为两个操作数的类型各不相同，所以MySQL将按数值方式对它们进行比较。因此，两个比较操作的结果都为真：

'2' < 11	→ 1
2 < '11'	→ 1

在对比较操作进行求值的时候，MySQL将根据以下原则对操作数进行相应的类型转换：

- 除<=>操作符以外，所有涉及NULL值的比较操作都将被求值为NULL。（“<=>”与“=”功能相当，但可以用来比较NULL值。表达式“NULL <=> NULL”将被求值为真。）
- 如果两个操作数都是字符串，它们之间的比较操作将按该类型的方式进行。对于二进制字符串，比较操作将一个字节一个字节地比较它们各个字节里的数值；对于非二进制字符串，比较操作将一个字符一个字符地按照它们在有关字符集里的排序顺序进行比较。如果两个字符串使用的字符集不同（这种情况可能出现在MySQL 4.1及以后的版本里），比较操作可能不会求值出有意义的结果。如果两个操作数一个是二进制字符串，另一个是非二进制字符串，它们之间的比较操作就将按二进制字符串方式进行。



- 如果两个操作数都是整数，它们之间的比较操作将按数值方式进行。
- 在MySQL 3.23.22及以后的版本里，比较操作中的十六进制常数将被视为数值。而在此之前，只要十六进制常数不是与数值进行比较，就都将被视为二进制字符串。
- 如果比较操作的两个操作数一个是TIMESTAMP或DATETIME值，另一个是一个常数，比较操作就会把它们都看做是TIMESTAMP值。这是为了使MySQL的比较操作能够与各种ODBC应用程序有更好的配合。
- 除上述各种情况外，比较操作中的操作数都将被视为浮点数。注意，字符串与数值之间的比较操作也包括在最后这种情况里。字符串将被转换为一个数值，如果字符串看起来不像是一个数字，转换结果就将是0。比如说，字符串'14.3'将被转换为浮点数14.3，但字符串'L4.3'却会被转换为0。

#### 1. 日期和时间值的解释规则

根据表达式中上下文的要求，MySQL会把字符串和数值自动转换为日期和时间值，反之亦然。在数值上下文里，日期和时间值将被自动转换为数值；在日期和时间上下文里，数值将被自动转换为日期和时间值。这种字符串或数值到日期和时间值的转换还会发生在：1) 当对某个日期和时间数据列进行赋值的时候；2) 使用了一个需要以日期和时间值为输入参数的函数的时候。在比较操作中，一般原则是把日期和时间值当做字符串来进行比较。

比如说，假设数据表mytbl里有一个名为data\_col的DATE数据列，那么下面几条语句将是等价的：

```
INSERT INTO mytbl SET date_col = '2004-04-13';
INSERT INTO mytbl SET date_col = '20040413';
INSERT INTO mytbl SET date_col = 20040413;
```

在下面的例子里，TO\_DAYS()函数的输入参数分别是三种不同类型的表达式，但它们都被解释为相同的值：

TO_DAYS('2004-04-10')	→ 732046
TO_DAYS('20040410')	→ 732046
TO_DAYS(20040410)	→ 732046

#### 2. 预查类型转换结果与强制类型转换

如果想预知在某个表达式的求值过程中都会发生什么样的类型转换，可以在mysql客户程序里用一条SELECT语句对这个表达式进行一次求值：

```
mysql> SELECT 0x41, 0x41 + 0;
+-----+-----+
| 0x41 | 0x41 + 0 |
+-----+-----+
| A    | 65       |
+-----+-----+
```

可以想像，为了写好这一章，我做了多少这样的查询！

对于那些用来删改数据记录的DELETE或UPDATE语句，提前预查表达式求值结果的做法有着重要的意义，因为必须确保这些语句只施加在想对之进行删改的数据行上。应该这样做：在

执行一条DELETE或UPDATE语句之前，先用一条SELECT语句对WHERE子句里的表达式做一下检查，看它选取出来的数据行是不是想要的东西。比如说，假设mytbl数据表里有一个名为char\_col的CHAR数据列，其中存放着以下数据：

```
'abc'
'def'
'00'
'ghi'
'jkl'
'00'
'mno'
```

那么，下面这条语句会把什么删除掉呢？

```
DELETE FROM mytbl WHERE char_col = 00;
```

本意可能只是想把包含着'00'值的那两个数据行删除掉，可这条DELETE语句的实际执行效果却是把所有的数据行全都给删除掉了——既让你大吃一惊，又让你后悔莫及！为什么会出现这样的结果呢？这全是MySQL的类型转换规则惹的祸。char\_col是一个字符串数据列，但上面这条DELETE语句里的00却因为没有放在引号中而被当做是一个数值。根据MySQL的类型转换规则，当把字符串与数值进行比较时，比较操作将把它的两个操作数都当做数值来对待。因此，在DELETE语句的执行过程中，char\_col数据列里的每个值将先被转换为数值，然后再与数值0进行比较。结果，'00'被理所当然地转换成了0，可其他字符串——因为看起来都不像是一个数值——也无一幸免地被转换成了0。于是，WHERE子句在每一个数据行上都成立，而DELETE语句也就顺理成章地把整个数据表给删除干净了。要是在执行DELETE语句之前先用一条SELECT语句对WHERE子句里的表达式进行检查，就肯定会注意到它选取出来的数据行超出了预想，你也就有机会避免刚才的悲剧了：

```
mysql> SELECT char_col FROM mytbl WHERE char_col = 00;
+-----+
| char_col |
+-----+
| 'abc'    |
| 'def'    |
| '00'     |
| 'ghi'    |
| 'jkl'    |
| '00'     |
| 'mno'    |
+-----+
```

如上所示，MySQL的类型转换规则的确会带来一些麻烦，但它并不是灾难的根源。在拿不准应该如何使用某项数据的时候，就应该利用MySQL的表达式求值机制或者MySQL提供的类型转换函数把它强行转换为所需要的类型：

- 给数据加上一个0或者加上一个0.0将把它强行转换为一个数值类型的值：

0x65	→ 'e'
0x65 + 0	→ 101
0x65 + 0.0	→ 101.0

- FLOOR()函数能够把浮点数强行转换为整数；给整数加上一个0.0将把它强行转换为浮点数：

FLOOR(13.3)	→ 13
13 + 0.0	→ 13.0

如果想得到四舍五入的效果，就要用ROUND()函数来代替FLOOR()函数。

- CONCAT()函数能够把任何类型的值转换为字符串：

14	→ 14
CONCAT(14)	→ '14'

在MySQL 4.0.2及以后的版本里，还可以用HEX()函数把数值转换为十六进制字符串：

HEX(255)	→ 'FF'
HEX(65535)	→ 'FFFF'

HEX()函数还可以把字符串转换为十六进制数字串，字符串里的每个字符将依次被转换为两个十六进制数字：

HEX('abc');	→ '616263'
-------------	------------

- ASCII()函数能够把字符转换为ASCII编码值：

'A'	→ 'A'
ASCII('A')	→ 65

如果想把ASCII编码反向转换为字符，就需要使用CHAR()函数：

CHAR(65)	→ 'A'
----------	-------

- DATE\_ADD()函数能够把字符串或者数值转换为日期和时间值：

20030101	→ 20030101
DATE_ADD(20030101, INTERVAL 0 DAY)	→ '2003-01-01'
'20030101'	→ '20030101'
DATE_ADD('20030101', INTERVAL 0 DAY)	→ '2003-01-01'

- 如果给日期和时间值加上一个0，就可以把它强行转换为数值：

CURDATE()	→ '2002-09-18'
CURDATE()+0	→ 20020918
CURTIME()	→ '12:05:41'
CURTIME()+0	→ 120541

- 在MySQL 4.1及以后的版本里，可以用CONVERT()函数把字符串从一个字符集转换到另一个字符集上去。在字符串的前面加上一个字符集标识符也能达到同样的效果：

'abc'	→ 'abc'
CONVERT('abc' USING ucs2)	→ '\0a\0b\0c'
CHARSET('abc')	→ 'latin1'
CHARSET(CONVERT('abc' USING ucs2))	→ 'ucs2'
CHARSET(_ucs2 'abc')	→ 'ucs2'

### 3. 超范围值或非法值的转换

MySQL对这类情况的基本处理原则是：垃圾进，垃圾出——如果事先没有对数据进行过检查，那所检索出来的信息就可能是一堆垃圾。话虽如此，MySQL还是会对那些超范围值或非法值尽量做些转换的，下面就是这方面的基本规则：

- 对于数值或TIME数据列，超出合法取值范围的数据值将被截断为距离最近的上限或下限值，MySQL将把截断后的结果值放到数据库里。
- 对于不是TIME类型的日期和时间数据列，超出合法取值范围的数据值将被转换为该类型的“零”值、NULL值或者其他值。（换句话说，其结果是不可预料的。）
- 对于不是ENUM或SET类型的字符串数据列，超长的字符串将被截短到该数据列的最大长度。对于ENUM或SET数据列，只有在该数据列的合法取值列表里出现过的值才是合法的，而未在其中出现过的值将全都被认为是非法的。如果试图把一个非法值赋值给某个ENUM数据列，就将导致MySQL把该ENUM数据列的出错成员（即对应于0值成员的那个空字符串）放入该数据列。如果试图把一个包含有非法子串的集合赋值给某个SET数据列，就将导致MySQL把剔除非法子串后留下来的成员放入该数据列。
- 对于日期和时间数据列，非法值将被转换为有关类型的“零”值（见表2-12）。

如果在ALTER TABLE、LOAD DATA、UPDATE、INSERT INTO……SELECT以及用来同时插入多个数据行的INSERT语句中发生了上述转换，MySQL就会给出一条相应的警告信息。在mysql客户程序里，这条警告信息将被直接显示在状态行上。利用编程语言提供的手段（比如MySQL C或PHP API里的函数mysql\_info()、Perl DBI API里的数据库连接属性mysql\_info等）也能获得这类信息。通过编程手段获得的信息是这类警告信息的计数值。





## 第3章 MySQL SQL语法及其使用

要想有效地与MySQL服务器进行通信，就必须熟练地掌握SQL语言，因为这种语言是MySQL的“母语”。例如，当使用mysql等客户程序的时候，其基本作用就是充当把SQL语句发送给MySQL服务器去执行的工具。此外，如果想利用某种程序设计语言所提供的MySQL接口来进行编程，也必须懂得如何使用SQL语言，因为那些接口函数也是充当向服务器发送SQL语句的工具，使你能够与MySQL服务器进行通信。

在本书第1章提供的教程里，我们对MySQL的许多功能进行了介绍。本章将在前述内容的基础上对在MySQL里实现的SQL语言的几个方面做更为详细的描述。将讨论MySQL数据库的组成元素，包括它们的命名规则和字母大小写情况的区分规则，还将介绍许多用来完成以下操作的重要的SQL语句：

- 数据库、数据表、索引的创建与删除操作。
- 获得关于数据库和数据表的信息。
- 使用关联（JOIN）、子选择（SUBSELECT）、联合（UNION）等方法来检索数据。
- 涉及多个数据表的删除和更改操作。
- 把多个语句当做一个单元来执行的事务处理机制。
- 建立外键关系。
- 使用FULLTEXT搜索引擎。

可以把MySQL的SQL语句归纳为几大类，表3-1列出了这几大类SQL语句中最有代表性的几种。在某些场合，SQL语句可以通过某个工具程序提供的命令行接口来发出。比如说，mysqlshow程序就允许通过命令行来完成SHOW操作。将在本章的有关内容里对这些工具程序加以注明。

表3-1里的部分语句没有在本章讨论，这是因为把它们安排到其他的章节去讨论将更合适一些。比如说，将把用来设置用户权限的管理性语句GRANT和REVOKE安排在第11章中进行讨论。第12章对MySQL数据库系统的各种访问权限以及它们所允许的各种操作做了集中的介绍。MySQL所实现的SQL语句的语法都可以在本书的附录D里查到。除此之外，还可以在MySQL *Reference Manual*（MySQL参考手册）里查到更多的信息，比如MySQL软件的新版本又对各有关语句做了哪些改进等。

MySQL并不是十全十美的，在本章的最后一节里，将对MySQL目前尚不具备的一些特征（比如触发器、存储过程（stored procedure）、视图（view）等等）进行介绍。这些功能尚未在MySQL中得到实现。那么，缺少这些功能是不是就意味着MySQL不是一个“真正”意义上的数据库系统呢？有些人的确是这样想的，但我要指出的是：虽然MySQL缺少了这样或那样的功能，但仍有许多人在使用它，对于许多甚至可以说是绝大多数应用程序而言，MySQL所缺少的这些功能并不重要。

表3-1 MySQL支持的SQL语句的分类

数据库的选取、创建、丢弃和变更
USE
CREATE DATABASE
DROP DATABASE
ALTER DATABASE
数据表及索引的创建、变更和丢弃
CREATE TABLE
DROP TABLE
CREATE INDEX
DROP INDEX
ALTER TABLE
获得关于数据库和数据表的信息
DESCRIBE
SHOW
从数据表检索信息
SELECT
UNION
事务处理
BEGIN
COMMIT
ROLLBACK
SET AUTOCOMMIT
对数据表里的信息进行修改
DELETE
INSERT
LOAD DATA
REPLACE
UPDATE
管理性语句
FLUSH
GRANT
REVOKE

此外，在开发者们的努力下，MySQL所欠缺的功能一直在不断地减少。在撰写本书第1版的时候，MySQL尚不具备事务处理、子选择、外键、引用完整性等功能。但从那时起，MySQL已经进行了相当大的改进，上面提到的这几项功能都已经添加到新版MySQL里了。我们相信，触发器、存储过程、视图等机制一定会出现在未来的MySQL版本里。

### 3.1 MySQL 的命名规则

几乎每一条SQL语句都需要以某种方式去引用一个数据库或者它的某些组成元素。本节将

对数据库、数据表、数据列、索引、别名等命名规则和引用规则进行讨论。名字中的字母通常还需要区分其大小写情况，所以还将讨论字母的大小写规则。

### 3.1.1 数据库组成元素的命名规则

在给数据库及其组成元素命名和通过这些名字来引用它们的时候，必须遵守MySQL的命名规则。首先，对用来构成名字的字符以及名字的长度是有限制的；其次，这些名字的具体写法还取决于它们将被用在什么样的上下文里。此外，MySQL允许在不同的命名模式下启动MySQL服务器，这也会对命名规则产生影响。

- **允许用在名字中的合法字符。**不带引号的名字可以由MySQL服务器默认字符集中的任意字母数字字符再加上“\_”（下划线字符）和“\$”（美元字符）构成。名字的第一个字符可以是命名规则所允许的任何一个合法字符——数字字符也可以作为名字的第一个字符。但MySQL不允许名字完全由数字组成，因为这将使它很难与数值区分开来。就数据库系统而言，MySQL允许以数字字符作为名字中第一个字符的做法有点不同寻常。如果使用了这样的名字，就要特别注意其中所包含的字母“E”和“e”，因为这两个字母会使名字产生二义性。举例来说，表达式“23e + 14”（加号两侧有空格）的意思是给名为“23e”的数据列加上一个数值14，“23e+14”（加号两侧没有空格）表达的又是什么意思呢？是与刚才同样的意思，还是一个用科学记数法表达的数字？

别名（alias）确实允许使用任意字符来构成，可以完全由数字字符组成，也可以包含空格或者其他一些特殊字符，但如果别名本身与某个SQL关键字完全相同的话，就应该把它放在一对单引号或双引号里。

从MySQL 3.23.6版本开始，还可以把名字放在一对反引号（```）里，这样，除反引号本身、ASCII 0、ASCII 255三个字符外，其他字符都允许用在这个名字里。当名字里包含有特殊字符或者名字是一个MySQL保留字时，这一点就十分有用。把名字放在一对引号里将允许其完全由数字字符构成，这在未加引号的名字里是不允许的。

不过，对于数据库和数据表的名字来说，即使把它们放在了引号里，也还有两个附加的约束条件：1）名字里不允许使用“.”（句点）字符，因为它是“`db_name.tbl_name`”和“`db_name.tbl_name.col_name`”表示法中的分隔符；2）名字里不允许使用UNIX和Windows的路径名分隔符“/”（斜线）或“\”（反斜线）。在数据库和数据表的名字里不允许使用这些分隔符的原因很简单——在硬盘上，每个数据库都将被表示为一个目录，而每个数据表则将至少被表示为一个文件。既然如此，不允许用在目录名和文件名里的非法字符当然也就不允许用在数据库和数据表的名字里了。至于不允许把UNIX路径名分隔符用在Windows系统上（反之亦然）的原因就更简单了，这是为了使运行在不同平台上的MySQL服务器能够更容易地使用对方的数据库和数据表。比如说，假如允许在Windows系统上的数据表名字中使用斜线字符的话，在UNIX系统上就无法使用这个数据表了，因为UNIX平台上的文件名是不允许使用斜线字符的。

- **名字的长度。**数据库、数据表、数据列和索引的名字最多允许64个字符，别名最多允许256个字符。

- **名字的限定符。**根据不同的上下文，可能需要给某些名字加上一些限定以使其表达的意义更加清晰。对于数据库的名字，可以像下面这样直接给出：

```
USE db_name;  
SHOW TABLES FROM db_name;
```

数据表的名字则有两种给出方式。首先，可以把数据库名和数据表名合在一起以构成该数据表的完整名称：

```
SHOW TABLES FROM db_name.tbl_name;  
SELECT * FROM db_name.tbl_name;
```

其次，如果单独给出数据表的名字，则表明它是当前的默认数据库里的一个数据表。比如说，如果sampdb是当前的默认数据库，那么下面两条语句将是等价的：

```
SELECT * FROM member;  
SELECT * FROM sampdb.member;
```

如果事先没有选定数据库，则给出一个不带数据库名的数据表名就将是非法的，因为MySQL服务器无法确定该数据表到底属于哪个数据库。

数据列的名字有全限定、部分限定和无限定三种给出方式。以全限定方式给出的数据列名（如`db_name.tbl_name.col_name`）由数据库名、数据表名和数据列名构成。部分限定的数据列名（如`tbl_name.col_name`）指的是给定数据表中的给定数据列。无限定数据列名（如`col_name`）指的是由上下文环境所确定的数据表里的给定数据列。请看下面两条查询语句，它们引用了同样的数据列名，但每条语句中的FROM子句表明了应该从哪一个数据表里选取相应的数据列：

```
SELECT last_name, first_name FROM president;  
SELECT last_name, first_name FROM members;
```

如果愿意，可以总是使用全限定名，而这样的名字也总是合法的，但通常没有必要这样做。假如已经用USE语句选定了了一个数据库，它就会成为当前的默认数据库并隐含在每一个用数据库名做出限定的数据表名字里。假如发出的SELECT语句只涉及一个数据表，该数据表就会隐含在这条语句所引用的每一个数据列名字里。只有在无法根据上下文来确定数据表或数据库的时候，才必须使用限定名。举例来说，如果某个查询语句涉及到了多个数据库里的多个数据表，那么，为了让MySQL知道要到哪些数据库里去找出这些数据表，就必须用`db_name.tbl_name`的格式来给出那些不在默认数据库中的数据表的名字。类似地，如果查询语句涉及到多个数据表而这些数据表里又有重名的数据列，为了清楚地表明所指的是哪一个数据表里的哪一个数据列，就必须用数据表的名字来限定各有关数据列的名字。

- **MySQL服务器的启动模式。**如果在启动MySQL服务器时使用了`--ansi`或`--sql-mode = ANSI_QUOTES`选项，就可以把名字放在一对双引号而不是一对反引号里（反引号仍可以使用）。



### 3.1.2 SQL语句对字母大小写的要求

SQL语句中的字母大小写规则根据语句的不同部分而变化，所指的具体事物和MySQL服务器运行在其上的计算机操作系统也会对此产生影响。

- **SQL关键字和函数名。**SQL关键字和函数名里的字母是不区分字母大小写的，它们既可以是\*\*大写字母\*\*，也可以是小写字母。下面这三条语句是等价的：

```
SELECT NOW();
select now();
sELeCt nOw();
```

- **数据库名和数据表名。**MySQL中的数据库和数据表是通过MySQL服务器主机上的文件系统里的目录和文件而实现的。因此，字母的大小写是否会对数据库名和数据表名产生影响将取决于MySQL服务器主机上的操作系统是如何对待文件名的。Windows文件名不区分字母的大小写，所以，运行在Windows操作系统上的MySQL服务器对数据库名和数据表名里的大小写字母也不加区分。UNIX文件名是区分字母大小写情况的，所以运行在UNIX操作系统上的MySQL服务器通常也要对数据库名和数据表名里的大小写字母加以区分。（有一个特例，即在Mac OS X下的HFS+文件系统里，对文件名里的大小写字母不加区分。）

如果打算在区分文件名字母大小写情况的MySQL服务器上创建数据库，并且可能会在将来把这个数据库移到某个不区分文件名字母大小写情况的服务器上去，就必须考虑字母的大小写问题。比如说，假设在UNIX服务器上创建了名为abc和ABC的两个数据表，这在UNIX操作系统中是两个不同的数据表名，但当把这两个数据表移到一台Windows机器上去的时候，就会遇到问题——因为Windows对名字里的字母大小写不加区分，所以Windows操作系统将无法区分abc和ABC数据表。要想避免因名字中的字母大小写方式而带来的问题，可以选定一种字母大小写方式（如小写字母），然后在创建数据库名和数据表名的时候坚持以这种方式来写出它们。这样，当在不同的服务器之间移动数据库的时候，名字中的字母大小写情况就不会构成问题了。这个问题的另一个解决办法是在启动MySQL服务器的时候对lower\_case\_table\_names变量进行设置。这个变量将在第10章里进行讨论。

- **数据列名和索引名。**在MySQL里，数据列名和索引名对字母的大小写不加以区分，下面这些查询语句具有相同的含义：

```
SELECT name FROM student;
SELECT NAME FROM student;
SELECT nAmE FROM student;
```

- **别名。**别名区分字母的大小写。可以用任意的字母大小写方式来写出别名（字母的大写、小写或是大小写混合），但当在查询语句中的其他地方用到这个别名时，请一定要用相同的大小写形式来写出这个别名。

不管系统是否区分数据库名和数据表名中的字母大小写情况，都应该在同一条查询语句里以前后一致的字母大小写形式来写出它们。这项原则不适用于SQL关键字、函数名、数据列名

和索引名——在查询语句中，它们的字母大小写形式可以任意变化。不过，如果坚持使用一种前后一致的字母大小写形式，所写出来的查询语句肯定会比随心所欲地“乱写”（比如SELECT Name FROM ...）要更容易阅读和理解。

## 3.2 数据库的选定、创建、丢弃和变更

MySQL提供了几个数据库级的语句：USE用来选定一个默认数据库；CREATR DATABASE用来创建数据库；DROP DATABASE用来丢弃（即删除）数据库；ALTER DATABASE用来改变数据库的全局特性。

### 3.2.1 数据库的选定

USE语句将选定一个数据库并把它当做指定MySQL服务器连接上的默认（当前）数据库：

```
USE db_name;
```

要想选定一个数据库，就必须相应地具备该数据库的一些访问权限，否则是无法选定它的。如果确实有访问该数据库的权限，那么即使没有选择数据库，只要用数据库名来限定数据表名，就可以使用其中的数据表。比如说，如果想在没有事先选定sampdb数据库的情况下检索该数据库里的president数据表内容，可以使用如下所示的查询语句：

```
SELECT * FROM sampdb.president;
```

显然，不带数据库限定符的数据表名用起来要更方便一些。

选定一个默认数据库并不意味着它将在连接持续期间内一直是默认数据库。只要具备足够的访问权限，就可以多次使用USE语句在多个数据库之间任意地来回切换。同时，选定一个数据库也并不意味着只能使用这个数据库里的数据表。即使已经把某个数据库选定为当前的默认数据库，也可以通过db\_name.tbl\_name形式的名字去访问其他数据库里的数据表。

当某个MySQL客户/服务器连接终止的时候，该服务器上的默认数据库概念也就不复存在了。换句话说，当再次连接上该服务器的时候，它并不会记得上一次是把哪个数据库选定为默认数据库的。事实上，这种想法本身就站不住脚——因为MySQL是多线程的，它允许同一个用户同时保持多条连接（当然，这些连接的起止时间不必相同）。在这种情况下，所谓“上一次选定的默认数据库”根本就无从谈起。

### 3.2.2 数据库的创建

创建数据库是一项非常简单的工作，只需要在CREATE DATABASE语句中给数据库起个名字就可以了：

```
CREATE DATABASE db_name;
```

执行数据库创建操作的先决条件是：所给出的数据库名必须是合法的，这个数据库不能已经存在，并且你必须有足够的权限去创建它。

### 3.2.3 数据库的丢弃

只要有足够的权限，丢弃（删除）数据库和创建数据库一样简单，使用如下语句即可：

```
DROP DATABASE db_name;
```

但要注意的是，千万不要随意使用DROP DATABASE语句，这条语句将会删掉数据库以及数据库里的数据表。一旦丢弃了一个数据库，这个数据库就永远消失了。换句话说，千万不要仅仅是为了看看这条语句有什么作用而去试验它。如果系统管理员一直在定期地对数据库进行备份，或许还能把数据库恢复回来。但我敢保证，假如你对系统管理员说：“天哪，我只不过是想看看使用DROP DATABASE语句会发生什么，可没想到……你能帮我把数据库找回来吗？”系统管理员肯定不会同情你。

注意：一个数据库就是MySQL数据目录里的一个子目录，这个子目录是存放数据表数据用的。有可能会发生这样的情况：你已经丢弃了一个数据库，但当使用SHOW DATABASE语句的时候，那个数据库的名字却依然会显示出来。导致这一现象的原因通常是因为那个数据库目录里还包含有一些与数据表无关的文件——DROP DATABASE语句不会删除这类文件，因而也就不删除那个数据库目录。这就意味着尽管其中已经没有数据表了，可数据库目录却依然存在。在这种情况下，如果真想丢弃那个数据库，就必须以手动方式去删除该数据库目录里遗留的文件以及该目录本身。

### 3.2.4 数据库的变更

在MySQL 4.1版本中，使用ALTER DATABASE语句可以改变数据库的全局特征或属性。就目前而言，数据库的全局特征还只有一个，即数据库的默认字符集：

```
ALTER DATABASE db_name DEFAULT CHARACTER SET charset;
```

*charset*应该是服务器所支持的某个字符集的名字，比如说latin1\_de或是sjis（如果想弄清楚服务器支持的字符集，可以使用SHOW CHARACTER SET语句）。*charset*也可以是DEFAULT，意思是数据库使用的是服务器级的默认字符集。在第2章里已经对字符集和字符集的级别进行了讨论。

数据库的属性都保存在相应的数据库目录里的db.opt文件里。

## 3.3 数据表的创建、丢弃、索引和变更

MySQL允许使用CREATE TABLE、DROP TABLE和ALTER TABLE语句来创建数据表、丢弃（删除）数据表以及改变数据表的结构。利用CREATE INDEX和DROP INDEX语句可以在现有的某个数据表里添加或删除索引。但在深入研究这些语句之前，先介绍MySQL所支持的各种数据表类型。

### 3.3.1 数据表类型

MySQL数据库支持多种数据表处理程序，每一种处理程序都相应地处理一种具有特定特性

集或是特征集的数据表类型。有多少种数据表类型可供选用的问题取决于MySQL版本、编译时的配置情况以及MySQL服务器启动时的选项设置情况。下表列出了MySQL目前能够支持的数据表类型处理程序以及它们最早出现于哪一个版本：

数据表类型	最早支持这一类型的MySQL版本号
ISAM	所有版本
MyISAM	3.23.0
MERGE	3.23.25
HEAP	3.23.0
BDB	3.23.17/3.23.34a
InnoDB	3.23.29/3.23.34a

BDB和InnoDB类型分别对应着两个版本号。第一个版本号指的是该数据表类型出现在二进制发行版本中的时候，第二个版本号指的是该数据表类型出现在源代码发行版本中的时候。MRG\_MyISAM和BerkeleyDB分别是MERGE和BDB的同义词。（InnoDB数据表类型在3.23.29到3.23.36版本中的名字是Innobase，以后的版本都以InnoDB为首选并把Innobase作为它的同义词。）

MySQL可以有多种配置方案，所以某给定版本的MySQL服务器有可能不支持该版本所能支持的全部数据表类型。第3.4节将对如何查知给定服务器到底都支持哪些数据表类型的问题进行介绍。第11.5.4节将对MySQL服务器的配置工作进行详细的讨论。

下面是对MySQL各种数据表类型的概述。

#### 1. ISAM数据表

ISAM处理程序负责管理那些使用索引化顺序访问方法的数据表。ISAM存储格式是最原始的MySQL数据表类型，也是3.23之前的版本仅有的一种数据表类型。从MySQL 3.23版本开始，MyISAM处理程序逐步取代了IASM处理程序，MyISAM数据表也逐渐成为主流——因为它们没有那么多的局限性。虽然现在还可以见到ISAM类型的数据表，但它已经非常过时了。随着时间的推移，ISAM类型很可能会彻底消失。（比如说，内嵌式MYSQL服务器现在已经不支持ISAM数据表了，这种数据表在MySQL 5里很可能彻底消失。）

#### 2. MyISAM数据表

MyISAM存储格式是MySQL 3.23及以后版本中默认的数据表类型——除非MySQL服务器有着其他的配置。这种数据表类型的优点是：

- 如果操作系统本身允许使用大尺寸文件，数据表的长度就能大于ISAM存储方式。
- 数据表的内容是以一种不依赖于具体机器的格式存储的。这意味着即使体系结构不同，也可以把数据表直接从一台机器上拷贝到另外一台机器上去。
- 与ISAM数据表相比较，MyISAM放宽了索引方面的几项限制。详细讨论参见第3.3.4节。
- 与ISAM格式相比较，MyISAM格式提供了更好的索引键字压缩效果。这两种格式在存储连续出现的、彼此相似的字符串索引值时均使用了压缩功能，但MyISAM还能对彼此相似的数字索引值进行压缩——因为数字值是按高字节在先的顺序存储的。（索引值中的低位字节变化较快，高位字节则相对稳定一些，所以比较容易对之进行压缩。）如果想激活对



数字键值的压缩功能,请在创建数据表时使用PACK\_KEYS = 1选项。

- 与其他类型的数据表相比,MyISAM在AUTO\_INCREMENT处理方面的能力更加强大。关于这部分的详细讨论可以参阅第2.3节。
- 作为对数据表完整性检查机制的一项改进,MyISAM数据表会在使用MySQL服务器或myisamchk程序对它进行检查时设置一个标志。MyISAM数据表还有一个用来表明数据表是否被正确关闭的标志,如果MySQL服务器异常关闭或者机器发生了崩溃,可以利用这个标志来判断需要对哪些数据表进行检查和修复。如果在启动MySQL服务器时使用了--myisam-recover选项,这种检查和修复将自动进行。
- MyISAM数据表处理程序支持通过使用FULLTEXT索引而进行的全文本搜索。

### 3. MERGE数据表

MERGE数据表是一种把多个MyISAM数据表组织为一个逻辑单元的方式。在查询MERGE数据表的时候,实际上就是在查询所有的子数据表。这样做的优点在于,对于单个的MyISAM数据表,可以使用超过文件系统所允许的数据表最大尺寸。

组成MERGE数据表的所有数据表都必须具有相同的结构。也就是说,各数据表里的数据列都必须有同样的名字、同样的类型和同样的先后顺序,索引也必须按同样的方式和同样的先后顺序来定义。压缩数据表和非压缩数据表允许混合在一起。(压缩数据表由myisampack程序生成,关于这部分内容请参阅附录E。)

MERGE数据表只能由同一个数据库里的数据表组成,不允许包含另外一个数据库里的数据表。

### 4. HEAP数据表

HEAP数据表存储格式使用的是驻留在内存里的数据表,而且各数据行的长度是固定的,这两个特征使得HEAP数据表的检索速度非常快。当MySQL服务器停止运行时,HEAP数据表也就消失了,从这个意义上讲,HEAP数据表是一种临时性的数据表。但它与我们使用CREATE TEMPORARY TABLE命令所创建的临时数据表是有区别的:HEAP数据表能够被其他客户程序看到。施加在HEAP数据表上的某些限制性规定使它们的处理工作更加简便、更加快捷:

- 索引仅用来完成操作符“=”及“<=>”所进行的比较操作。由于使用了散列化索引技术,所以它的“等于”比较操作的执行速度非常快,但在“<”或“>”之类的操作符所进行的区间式搜索中,它的速度就比较慢了。正是因为这个原因,HEAP数据表上的ORDER BY子句也没有使用索引。
- 在MySQL 4.0.2之前的版本中,在被索引的数据列里不允许有NULL值。
- 在MySQL 4.1之前的版本中,不允许使用AUTO\_INCREMENT数据列。
- 不允许使用BLOB和TEXT类型的数据列。因为数据行是用固定长度的格式存储的,所以不能使用诸如BLOB和TEXT等可变长度的数据列类型。VARCHAR类型的数据列尽管可以使用,但在内部,它实际上是被作为CHAR类型来处理的。

### 5. BDB数据表

BDB数据表由Sleepycat公司开发的Berkeley DB数据库处理程序管理。BDB处理程序具有如下功能:

- 支持事务处理机制，即允许使用提交（COMMIT）和回滚（ROLLBACK）操作。
- 崩溃后自动恢复。
- 在同时包含有检索和更改命令的混合查询条件下，可以页面级锁定，也就是具有良好的并发性能。

#### 6. InnoDB数据表

InnoDB数据表是最近才添加到MySQL里的数据表类型。它们由Innobase Oy公司开发的InnoDB处理程序管理。InnoDB处理程序具有如下功能：

- 支持事务处理机制，即允许使用提交（COMMIT）和回滚（ROLLBACK）操作。
- 崩溃后自动恢复。
- 外键支持，包括级联删除。
- 在同时包含有检索和更改命令的混合查询条件下，可以数据行级锁定，也就是具有良好的并发性能。
- InnoDB数据表是在一个分开的数据表空间里被管理的，而不是像其他的数据表类型那样，使用特定的数据表文件来进行管理。数据表空间可以由多个文件组成，并且可以包含原始分区。事实上，InnoDB处理程序是将数据表空间作为一个虚拟的文件系统来对待的，在这个虚拟的文件系统里，InnoDB处理程序管理着所有InnoDB数据表的内容。
- 通过使用多个文件或是数据表空间里的原始分区，数据表尺寸可以超过文件系统对单个文件所允许的最大尺寸。

#### 7. 数据表在磁盘上的表示方法

每一个数据表，无论是哪一种类型，都可以在磁盘上被表示为一个包含有该数据表的格式（即它的定义）的文件。这个文件的基本名与该数据表的名字相同，扩展名是.frm。就大多数数据表类型而言，数据表的内容通常都被保存在其他一些文件里，这些文件与该数据表有着惟一的对应关系。但HEAP和InnoDB类型的数据表是特例，对于这两种数据表，.frm文件是与给定数据表相关联的惟一的文件。（HEAP数据表的内容存储在内存里。InnoDB数据表的内容是与其他InnoDB数据表一起被存放在InnoDB数据表空间里的，各有关数据表没有自己独立存在的文件。）各种数据表类型使用带有如下扩展名的文件：

数据表类型	磁盘上的文件
ISAM	.frm（定义）、.ISD（数据）、.ISM（索引）
MyISAM	.frm（定义）、.MYD（数据）、.MYI（索引）
MERGE	.frm（定义）、.MRG（子MyISAM数据表名字清单）
HEAP	.frm（定义）
BDB	.frm（定义）、.db（数据和索引）
InnoDB	.frm（定义）

对于任意给定的数据表，与它有关的文件都放在该数据表所在数据库的目录里。

#### 8. 数据表类型的可移植性

可以用mysqldump程序把数据表的内容导出到一个文本文件里，再把这个文本文件移到运行有MySQL服务器的其他机器上去并通过加载该文本文件的办法来重建数据表。从这个意义上讲，

任何一个数据表都是可移植的。但在这一小节里，“可移植性”的意思是可以直接把数据表文件拷贝到磁盘上，再把磁盘里的文件直接拷贝到另一台MySQL服务器主机的某个数据库目录里，而那台主机上的MySQL服务器就能直接使用该数据表了。当然，这个定义不适用于HEAP数据表，因为它们的内容是存放在内存而不是磁盘里的。对于其他类型的数据表，有些是可移植的，而有些则是不可移植的。

- ISAM数据表在磁盘上的存储格式与具体的机器有关，所以它们只有在具有相同硬件特性的机器之间才具备可移植性。
- BDB数据表是不可移植的，这是因为BDB数据表的存储位置被编码在它的.db文件里。也就是说，BDB数据表与创建它们时所使用的机器上的文件系统的某个特定位置有关。（实际上，就BDB数据表的可移植性来说，这是一个比较保守的观点。我本人曾对BDB数据表进行过各种可移植性方面的尝试，比如说把它们从一个数据库目录拷贝到另一个数据库目录、把它们重新命名为不同的基本名等等，我没有发现这样做有什么问题。但还是先用mysqldump程序来导出它们、再通过加载导出文件的办法把它们重新创建在目标机器上的做法最保险。）
- MyISAM数据表和InnoDB数据表的存储格式与具体的机器无关，所以它们是可移植的。只要两台MySQL主机的处理器都使用二进制补码来表示整数且使用的都是IEEE浮点格式，就能在它们之间自由地对数据表进行移动。这两个条件通常不会对数据表的可移植性构成真正的问题——除非有一种很特殊的机器。一般说来，在实际工作中，这两种数据表在可移植性方面的问题主要出在所使用的是一种为专用设备而建立的内嵌式MySQL服务器的场合，因为专用设备的处理器通常会有一些较为特殊的操作特性。
- MERGE数据表的可移植性取决于组成该MERGE数据表的各个MyISAM文件，只要它们是可移植的，MERGE数据表就是可移植的。

从本质上讲，MyISAM数据表和InnoDB数据表对可移植性的要求是：它们根本不包含任何浮点数据列，或者两台MySQL服务器主机所使用的浮点存储格式完全相同。这里所说的“浮点”指的是FLOAT和DOUBLE，以字符串格式存储的DECIMAL数据列是可移植的。

需要注意的是，InnoDB数据表的可移植性取决于它所在的数据表空间而不是数据表本身。InnoDB处理程序把所有InnoDB数据表的内容都存储在同一个数据表空间里，而不是存储在它们各自的数据表文件里。因此，InnoDB数据表是否具备可移植性取决于它所在的InnoDB数据表空间文件能否被移植，而不取决于各个InnoDB数据表本身能否被移植。这意味着只有InnoDB数据表空间里的每一个InnoDB数据表都满足刚才提到的对浮点数据列的要求，这些InnoDB数据表才是可移植的。

除各种数据表类型本身对可移植性的要求以外，还必须注意这样一个问题：只有在MySQL服务器被“干净地”关闭之后才可以把数据表或数据表空间文件拷贝到另外一台机器上去。如果MySQL服务器的关机操作不“干净”，所拷贝出来的数据表副本就可能不完整，可能还需要对它们进行修复或者把数据表处理程序的日志文件里尚未提交或回滚的操作完成之后才能得到最新、最完备的数据表副本。

类似地，如果MySQL服务器正在运行并正在修改着某个数据表的内容，因为修改结果尚未

写入磁盘，所以与该数据表相关联的文件不可能产生有用的数据表副本。因此，只要MySQL服务器仍在运行，就不应该去拷贝它正在修改的数据表——拷贝操作应该跳过那些正在被修改的数据表，关于这一问题的详细讨论请参阅第13章。

### 3.3.2 数据表的创建

要想创建数据表，就要用到CREATE TABLE语句。这个语句的选项特别多，所以其完整语法非常复杂，但它在实际工作中的使用方法却是非常简单的。比如说，第1章中所使用的所有CREATE TABLE语句就不复杂。CREATE TABLE语句的基本形式不会给你带来多大的麻烦。

在最简单的情况下，CREATE TABLE语句只需你给出一个数据表名和其中的数据列清单，如下所示：

```
CREATE TABLE mytbl
(
    name    CHAR(20),
    age     INT NOT NULL,
    weight  INT,
    sex     ENUM('F','M')
);
```

除了组成数据表的数据列以外，还可以在创建数据表的时候就定义好如何对数据表进行索引。另一种做法是在创建数据表的时候不对它进行索引，等以后再把索引添加进去。（如果在开始对某个MyISAM或ISAM数据表进行查询之前还需要往其中添加大量的数据，在创建该数据表时先不创建索引就是一个很好的策略。对于这两种类型的数据表，每插入一个数据行就更新一次索引的做法要比先把有关数据全部加载完毕再创建索引的做法慢得多。）

在第1章中，介绍了CREATE TABLE语句的基本语法；在第2章里，又讨论了如何给出数据列的定义。在这里，假设你们都已经学习过前两章的内容，所以就不再重复已经讲过的东西了。在本小节里，将着重介绍CREATE TABLE语句自MySQL 3.23版本才开始新增的一些扩展机制，利用这些机制，可以更灵活地创建数据表：

- 一些能够改变其存储特性的数据表选项。
- 如果数据表已经存在怎么办？
- 如何创建和使用临时数据表（即那些在本次客户会话结束时被自动丢弃的数据表）？
- 怎样才能使用SELECT语句的查询结果来创建数据表？
- MERGE数据表的使用。

#### 1. 数据表选项

从MySQL 3.23版本开始，可以在CREATE TABLE语句的右括号后面添加一些数据表选项来改变数据表的存储特性。比如说，在MySQL 3.23之前的版本里，因为只有ISAM类型可供使用，所以只能创建出这种类型的数据表来。但在3.23及以后的版本里，可以用一个TYPE = *tbl\_type*选项来明确地对数据表的类型进行设定。举例来说，要创建一个HEAP数据表或是一个InnoDB数据表，使用如下所示的语句（数据表类型名不需要区分字母的大小写）：



```
CREATE TABLE mytbl ( ... ) TYPE = HEAP;
CREATE TABLE mytbl ( ... ) TYPE = INNODB;
```

如果没有指定数据表类型，MySQL服务器将使用默认的数据表类型来创建数据表。如果既没有在重新配置MySQL服务器的时候给它另行设定一个默认的数据表类型，也没有在启动MySQL服务器的时候使用--default-table-type选项，MySQL服务器就将以MyISAM作为默认的数据表类型。此外，在创建某个数据表的时候，如果所给出的数据表类型名没有错误但MySQL服务器里没有相应的处理程序，MySQL也将使用默认的数据表类型来创建该数据表。如果所给出的是一个非法的数据表类型名，CREATE TABLE语句将报告出错。

还有其他一些数据表选项可供选用，其中大多数都只适用于特定数据表类型。比如说，与HEAP数据表一起使用的MIN\_ROWS = *n*选项将使HEAP处理程序对内存进行优化：

```
CREATE TABLE mytbl ( ... ) TYPE = HEAP MIN_ROWS = 10000;
```

如果处理程序认为MIN\_ROWS的值过大，就会使用较大的内存块来分配内存以避免因需要进行多次内存分配调用而产生的额外开销。

数据表选项的完整清单可以在附录D中的CREATE TABLE条目处查到。

如果数据表已经存在，还可以用各种数据表选项配合ALTER TABLE语句来改变它的当前特性。比如说，下面这条语句将把mytbl数据表的当前类型改变为InnoDB：

```
ALTER TABLE mytbl TYPE = INNODB;
```

能否对数据表的类型进行转换取决于新、旧两种数据表类型之间的功能是否兼容。比如说，如果某个MyISAM数据表里包含有BLOB数据列，就不能把它转换为HEAP格式，因为HEAP数据表根本就不支持BLOB数据列。

## 2. 如果数据表已经存在怎么办

如果只想在某个数据表尚不存在的情况下才创建它，就需要使用CREATE TABLE IF NOT EXISTS语句，MySQL 3.23.0及以后的版本都支持这种做法。如果把这条语句用在应用程序里，应用程序就用不着关心它将要用到的数据表是否已经存在了——它会一直往下进行，无论各有关数据表是否存在都不会出错。对于那些供mysql客户程序去调用执行的批作业脚本来说，IF NOT EXISTS修饰符就更有用了。在此上下文里，平常的CREATE TABLE语句往往不能正确工作：在第一次运行的时候，批作业能够创建出数据表来；但在第二次运行的时候，因为数据表已经存在，有关脚本将报告出错并因此而停止执行。如果使用IF NOT EXISTS修饰符，就不会出现上述问题了：在第一次运行的时候，批作业像刚才那样创建出了数据表；在第二次及以后运行的时候，批作业将默默地跳过那些数据表创建操作，有关脚本既不会报告出错，更不会因此而停止执行——批作业将按数据表创建成功的路线执行下去。

## 3. 临时数据表

可以使用CREATE TEMPORARY TABLE语句来创建临时数据表，“临时”是指这类数据表在本次会话结束时将自动消失。这种数据表的方便之处在于，不必使用一条DROP TABLE语句去丢弃它——即使本次会话因某种原因而异常终止，在这次会话过程中创建的临时数据表也不会挂在内存里。比如说，假如在用mysql程序调用执行的某个批处理脚本里有一个事先写好的查

询，如果决定不等它完成就退出，就可以毫无顾忌地在该脚本的执行过程中杀死有关的进程，并且MySQL服务器将自动地把该脚本创建的临时数据表全都删掉。

临时数据表仅对创建它的客户程序可见。临时数据表的名字允许与一个已经存在的永久数据表的名字相同。这样做不会发生错误，已经存在的永久数据表也不会遭到破坏。在临时数据表存在的时候，永久数据表将会隐藏起来（不可访问）。假定在sampdb数据库中创建了一个名为member的临时数据表，那么原来的member数据表就会隐藏起来，这时引用的member数据表就是临时数据表。此时，当发出DROP TABLE member语句的时候，被删除的将是临时数据表，而原来的member数据表将会“重新出现”。如果只是简单地断开了与服务器的连接而没有删除临时数据表，MySQL服务器将自动地把它删除掉。等下次再与服务器连接的时候，原来的member数据表就又是可见的了。（如果更改了临时数据表的名字，与之同名的永久数据表将会重新出现，但如果临时数据表改名后恰巧与另一个永久数据表的名字相同，则那个永久数据表就将被隐藏起来。）

这种数据表名的隐藏机制只支持一个临时数据表，也就是说，不能用同一个名字创建出两个临时数据表来。

在创建TEMPORARY数据表的时候，还可以用TYPE选项为它指定一个存储格式（即数据表类型）。（注意：在MySQL 3.23.54之前的版本里，MERGE数据表不允许是TEMPORARY。）

在MySQL 3.23.2之前的版本里，不存在TEMPORARY数据表的概念，也没有真正意义上的临时数据表——只能在自己心里记住哪几个数据表是供临时使用的。这种意义上的“临时”数据表必须由自己负责删除，如果忘记了，它们将一直挂在那儿直到你注意到它们并明确地删除它们为止。需要特别提醒大家注意的是：如果应用程序创建一个这样的“临时”数据表但在丢弃该数据表之前由于意外原因而退出了执行，这个“临时”数据表将一直挂在内存里，直到你采取了必要的措施为止。

#### 4. 从SELECT语句的查询结果创建数据表

关系数据库的一个重要概念就是一切事物都可以表示成一个由数据行和数据列构成的表，而每个SELECT语句的查询结果也是一个由数据行和数据列构成的数据表。在大多数情况下，作为SELECT查询结果的“数据表”只不过是有关数据行和数据列在显示器屏幕上的映像，它们将从显示器屏幕的底部上卷到顶部，然后消失不见。但在某些场合，可能需要把某次查询的结果保存为一个真正的数据表以供今后使用。

从MySQL 3.23.0版本开始，可以很轻松地做到这一点。CREATE TABLE ... SELECT语句能够即时创建一个数据表并把SELECT语句的查询结果存放到这个数据表里去。这两项工作是在同一个操作里完成的，根本用不着事先对打算检索的数据列的数据类型进行定义。这可太方便了：快速地创建了一个数据表，其中的数据也都填好了，而且立刻就能把它用在后面的查询里。我们来看几个例子。下面这条语句将创建一个名为student\_f的新数据表，其中是student数据表里所有女学生的信息：

```
CREATE TABLE student_f SELECT * FROM student WHERE sex = 'f';
```

要想复制整个数据表，省略掉WHERE子句就行了：

```
CREATE TABLE new_tbl_name SELECT * FROM tbl_name;
```

要想创建一个无内容的数据表副本，使用一个求值结果恒为假的WHERE子句就行了：

```
CREATE TABLE new_tbl_name SELECT * FROM tbl_name WHERE 0;
```

当准备使用LOAD DATA语句把一个数据文件加载到原始数据表里却又不能肯定所使用的选项是否能正确地设定数据格式的时候，就应该先像上面这样创建一个无内容的数据表副本。要知道，如果第一次使用的选项不正确，加载到原始数据表里的数据记录就会被弄得乱七八糟！有了原始数据表的空白副本，就能毫无顾虑地利用LOAD DATA语句的有关选项去设定数据列和数据行分隔符，直到所设定的分隔符能够把输入记录正确地分隔为各个字段为止。等满意之后，就可以把数据文件加载到原始数据表里去了。完成这项工作有两个办法：一是仍利用LOAD DATA语句但要使用试验满意的分隔符把数据加载到原始数据表里；二是从副本（它已经在试验过程中把数据加载好了）里把数据直接拷贝过去，如下所示：

```
INSERT INTO orig_tbl SELECT * FROM copy_tbl;
```

如果把CREATE TEMPORARY TABLE和SELECT语句结合起来，就可以把数据表的内容检索到它本身的一个临时副本里：

```
CREATE TEMPORARY TABLE mytbl SELECT * FROM mytbl;
```

这将允许修改数据表的内容而不会影响到原始的数据表，在想试着用一些查询来修改数据表内容却又不想改变原始数据表的时候将是非常有用的：当使用一个预先写好的脚本而这个脚本又使用着原始数据表的名字时，不需要编辑这个脚本以改变其中的数据表名字，只要在脚本的开头加上CREATE TEMPORARY TABLE语句就可以了。脚本将为各有关数据表创建一个临时副本并在其上进行操作，等脚本结束时，MySQL服务器会自动删除副本。（但这里需要注意这样一个问题：有些客户程序（例如mysql）在与服务器的连接掉线时会自动尝试重新建立与服务器的连接，如果这种情况发生在你正使用临时数据表进行工作的时候，临时数据表将在连接掉线时被删除掉，重建连接后再执行的查询将作用于原始数据表。如果你的网络连接不太可靠，就要特别注意这一问题。）

如果想为某个数据表创建一个无内容的空白副本，可以在发出CREATE TEMPORARY TABLE...SELECT语句的时候给出一个求值结果恒为假的WHERE子句：

```
CREATE TEMPORARY TABLE mytbl SELECT * FROM mytbl WHERE 0;
```

用SELECT语句的查询结果即时创建一个数据表是一项非常有用的功能。但在这样做的时候，一定要注意以下几个问题：

首先，在使用CREATE TABLE...SELECT语句的时候，应该使用一些必要的别名以提供有意义的数据列名。当通过把数据选取到数据表里的方法来创建数据表的时候，数据列的名字将沿用所选择的数据列。假如某个数据列是某个表达式的计算结果，该数据列的“名字”就将是那个表达式的文本。在MySQL 3.23.6之前的版本里，下面的语句将执行失败，因为表达式不能作为数据列的合法名字：

```
mysql> CREATE TABLE mytbl SELECT PI();
ERROR 1166: Incorrect column name 'PI()'
```

MySQL 3.23.6及以后的版本放宽了数据列的命名规则，所以下面这条语句将能够创建出一

个数据表，但其中的数据列却会有一个不同寻常的名字，如下所示：

```
mysql> CREATE TABLE mytbl SELECT PI();
mysql> SELECT * FROM mytbl;
+-----+
| PI()   |
+-----+
| 3.141593 |
+-----+
```

这并不是一个令人非常满意的结果，因为那个数据列只能通过把其名字放在反引号里才能引用，如下所示：

```
mysql> SELECT 'PI()' FROM mytbl;
+-----+
| PI()   |
+-----+
| 3.141593 |
+-----+
```

因此，如果在SELECT语句里使用了表达式，就应该给它起一个别名以便能够相对简单地引用有关的数据列，如下所示：

```
mysql> CREATE TABLE mytbl SELECT PI() AS mycol;
mysql> SELECT mycol FROM mytbl;
+-----+
| mycol   |
+-----+
| 3.141593 |
+-----+
```

如果打算从不同的数据表里选取一些名字相同的数据列，也会遇到同样的问题。假如数据表t1和数据表t2中都有一个数据列c，想把这两个数据表的数据行的所有组合创建一个数据表，使用下面的语句将会失败，因为这条语句试图创建一个有两个数据列的名字都是c的数据表：

```
mysql> CREATE TABLE t3 SELECT * FROM t1, t2;
ERROR 1060: Duplicate column name 'c'
```

可以利用别名为新数据表提供两个彼此不同的数据列名字：

```
mysql> CREATE TABLE t3 SELECT t1.c AS c1, t2.c AS c2 FROM t1, t2;
```

另一个需要注意的事情是，如果原始数据表的某些特征没有体现在选取出来的数据里，这些特征也将无法体现在新数据表的结构定义里。比如说，通过选取数据而创建出来的数据表不会把原始数据表里的索引也自动地复制到新数据表里，因为结果集本身并没有被索引。类似地，数据列属性（比如AUTO\_INCREMENT或默认值）通常也不会被带入到新数据表里（新版本要比老版本做得好一些）。在某些场合，可以通过调用CAST()函数把原始数据表的某些特定属性强行复制到新数据表里，CAST()函数最早出现于MySQL 4.0.2版本。下面这条CREATE TABLE...SELECT语句将使SELECT生成的数据列具备INT UNSIGNED、DATE和CHAR BINARY属性。



可以用一条DESCRIBE语句来验证这一点:

```
mysql> CREATE TABLE mytbl SELECT
-> CAST(1 AS UNSIGNED) AS i,
-> CAST(CURDATE() AS DATE) AS d,
-> CAST('Hello, world' AS BINARY) AS c;
```

```
mysql> DESCRIBE mytbl;
```

Field	Type	Null	Key	Default	Extra
i	int(1) unsigned			0	
d	date			0000-00-00	
c	char(12) binary				

还可以把CAST()函数应用到从其他数据表检索出来的数据列值上。能够被CAST()函数转换的类型包括: BINARY (二进制字符串)、DATE、DATETIME、TIME、SIGNED、SIGNED INTEGER、UNSIGNED以及UNSIGNED INTEGER。

从MySQL 4.1开始, 还可以通过明确地对新数据表里的数据列进行定义来提供更多的类型信息, 新数据表里的数据列与在SELECT语句里选取的数据列是通过名字而相互对应的, 只要为被选取的数据列提供了必要的别名, 就可以使新、旧数据表里的数据列得到正确的匹配:

```
mysql> CREATE TABLE mytbl (i INT UNSIGNED, d DATE, c CHAR(20) BINARY)
-> SELECT
-> 1 AS i,
-> CURDATE() AS d,
-> 'Hello, world' AS c;
```

```
mysql> DESCRIBE mytbl;
```

Field	Type	Null	Key	Default	Extra
i	int(10) unsigned	YES		NULL	
d	date	YES		NULL	
c	char(20) binary	YES		NULL	

注意, 这种方法允许新创建出来的字符数据列与结果集中最长的值的宽度不同。还要注意, 在这个例子里, 数据列的Null和Default属性与前一个例子是不同的。对于这些属性, 应该视具体情况而明确地做出必要的定义。

MySQL 3.23之前的版本没有提供CREATE TABLE ... SELECT语句。如果想把SELECT语句的查询结果存放到一个数据表里供今后的查询使用, 就必须提前做出特殊的安排:

1) 先用DESCRIBE或SHOW COLUMNS查询来了解源数据表 (即想从中选取数据列的数据表) 的数据列类型。

2) 明确地发出一条CREATE TABLE语句来创建目标数据表 (即将用来存放SELECT结果的数据表), 它应该对将用SELECT语句检索的数据列的名字和类型做出准确的定义。

3) 创建好目标数据表后, 用INSERT INTO ... SELECT查询来检索数据并把结果插入到新数据表里。

很显然, 与CREATE TABLE ... SELECT语句相比, 上面的过程要复杂得多。

#### 5. 使用MERGE数据表

MERGE数据表类型最早出现于MySQL 3.23.25版本, 在此后的版本里都能使用。它提供了一种把一组数据表当做一个逻辑单元来同时进行查询的机制。根据第3.3.1节中的介绍, 这种MERGE机制可以施加到一组结构完全相同的MyISAM数据表上。比如说, 假设有一些用来充当操作日志的数据表(这些数据表分别保存着不同年份的操作记录), 它们都有如下所示的同样的定义(数据表名字里的“CCYY”代表世纪和年份):

```
CREATE TABLE log_CCYY
(
    dt    DATETIME NOT NULL,
    info  VARCHAR(100) NOT NULL,
    INDEX (dt)
) TYPE = MYISAM;
```

再假设当前的日志数据表集合里包含着log\_1999、log\_2000、log\_2001、log\_2002以及log\_2003, 可以建立一个如下所示的MERGE数据表来映射这些日志数据表:

```
CREATE TABLE log_all
(
    dt    DATETIME NOT NULL,
    info  VARCHAR(100) NOT NULL,
    INDEX (dt)
) TYPE = MERGE UNION = (log_1999, log_2000, log_2001, log_2002, log_2003);
```

TYPE选项的值必须是MERGE, 而UNION选项列出了将被收录在MERGE数据表里的所有数据表。创建出这个MERGE数据表之后, 就可以像对待其他任何数据表那样去查询之, 只不过发出的查询命令将在组成该MERGE数据表的各个日志数据表上同时进行。例如, 下面这个查询将得到全体日志数据表里总共有多少个数据行:

```
SELECT COUNT(*) FROM log_all;
```

下面这个查询将得到每年(即每个日志数据表里)有多少个日志项:

```
SELECT YEAR(dt) AS y, COUNT(*) AS entries FROM log_all GROUP BY y;
```

除了能够很方便地用一条查询命令(而不需要使用多个查询命令)对多个数据表进行查询外, MERGE数据表还有以下一些优点:

- MERGE数据表可以用来创建这样一种逻辑实体: 它的长度允许超过单个MyISAM数据表所允许的最大尺寸。
- 构成MERGE数据表的子数据表允许是压缩数据表。比如说, 假设在某个给定的年份结束以后不会再往相应的日志文件里添加新的内容, 就可以用myisampack工具程序来压缩它以节省空间。这种压缩不会影响MERGE数据表的正常使用。
- MERGE数据表上的操作与UNION操作的情况非常相似。但因为MySQL 4之前的版本根本

不具备UNION机制，所以，在某些场合，可能需要用MERGE数据表来实现UNION机制。

MERGE数据表支持DELETE和UPDATE操作，但INSERT操作的情况就要复杂一些，这是因为MySQL需要知道应该把新记录插入到哪一个数据表里。从MySQL 4.0.0版本开始，MERGE数据表的定义语法里增加了一个名为INSERT\_METHOD的选项，这个选项的可取值NO、FIRST或LAST分别表明该MERGE数据表不允许INSERT操作、新记录将被插入到UNION选项所列举的第一个或者最后一个数据表里去。比如说，下面的定义将使MERGE数据表log\_all上的INSERT语句把新记录插入到日志数据表log\_2003里去，因为它是UNION选项所列举的最后一个数据表：

```
CREATE TABLE log_all
(
    dt      DATETIME NOT NULL,
    info    VARCHAR(100) NOT NULL,
    INDEX (dt)
) TYPE = MERGE UNION = (log_1999, log_2000, log_2001, log_2002, log_2003)
INSERT_METHOD = LAST;
```

### 3.3.3 数据表的丢弃

丢弃数据表要比创建它简单得多，因为用不着在丢弃数据表的时候对它的内容进行任何设定，只要发出一条下面这样的语句就行了：

```
DROP TABLE tbl_name;
```

MySQL对DROP TABLE语句做了一些很有用的扩展。首先，可以用一条DROP TABLE语句丢弃多个数据表，只要在同一条DROP TABLE语句里把它们都列举出来就行了：

```
DROP TABLE tbl_name1, tbl_name2, ... ;
```

其次，如果不能肯定某个数据表是否存在，但如果它存在的话就丢弃之，可以在DROP TABLE语句里加上IF EXISTS关键字。这样，即使在DROP TABLE语句里给出了一个实际并不存在的数据表，MySQL也不会报告出错：

```
DROP TABLE IF EXISTS tbl_name;
```

IF EXISTS特别适合用来编写将由mysql客户程序调用执行的脚本。在默认情况下，mysql客户程序将在执行出错（比如试图删除一个并不存在的数据表）时退出执行。我们来看一个例子。假设有一个用来创建数据表的安装脚本，其他脚本将对它所创建的数据表进行处理。在这种情况下，当然希望这个安装脚本能够在一个“干净”的环境里开始执行。如果在这个安装脚本的开头使用的是普通的DROP TABLE语句，那么，如果有关的数据表尚未被创建出来，它就会执行出错；可如果还使用了IF EXISTS关键字，就不会出现这样的问题了——数据表如果存在，就会被丢弃掉；如果不存在，脚本也将继续执行。

### 3.3.4 数据表的索引

索引是加快数据表内容访问操作的基本手段，对于那些涉及多个数据表的查询更是如此。

这是一个非常重要的话题，所以将在第4章里对为什么要使用查询、它们是如何工作的、怎样才能利用索引来优化查询等问题做专门的讨论。这一小节的主要内容是各种数据表类型的索引特性以及用来创建和丢弃索引的语法。

### 1. 各数据表类型的索引特性

MySQL为索引的构建工作提供了一些很灵活的办法：

- 可以索引单个的数据列，也可以根据数据列的各种组合来构建复合索引。
- 索引既可以包含重复的键值，也可以只包含惟一的索引值。
- 可以为同一个数据表创建多个索引，这样，根据不同数据列都能快速检索出某项数据。
- 对于不是ENUM和SET类型的字符串数据列，可以只对它们的一个前缀（即最左边的前 $n$ 个字符）进行索引。（事实上，对于BLOB和TEXT类型的数据列，如果不指定前缀的长度，是无法建立索引的。）前缀最多可以有255字节。一般来说，如果数据列的前 $n$ 个字节已经是非常惟一了，这种前缀形式的索引不仅不会导致系统性能的降低，相反，它往往会使系统性能得到很大的改善。对数据列的前缀而不是整个数据列进行索引能够有效地压缩索引的尺寸并提高其访问速度。

不过，并非所有的数据表都具备所有的索引特性。下表给出了各种数据表类型的索引特性。（这个表格没有收录MERGE类型，因为MERGE数据表是由一组MyISAM数据表创建出来的，所以具有与MyISAM数据表类似的索引特性。）

索引特性	ISAM	MyISAM	HEAP	BDB	InnoDB
是否允许NULL值	否	是	从4.0.2版本开始	是	是
每个数据列最多可有多少个索引	16	16	16	16	16
每个数据表最多可有多少个索引	16	32	32	31	32
索引行的最大长度（字节）	256	500	500	500 / 1024	500 / 1024
是否允许对数据列前缀进行索引	是	是	是	是	否
是否允许对BLOB / TEXT数据列进行索引	否	是（最大 255字节）	否	是（最大 255字节）	否

在“索引行的最大长度（字节）”一栏中，BDB数据表和InnoDB数据表都有两个数值。对于这两种类型的数据表来说，在4.0.3版本以前，是500字节，在此之后，是1024字节。

上表还反映出了为什么人们更青睐MyISAM存储格式而不是ISAM存储格式。MyISAM放宽了ISAM数据表对索引的某些限制，比如说，对于MyISAM数据表，可以索引含有NULL值的数据列、可以索引BLOB和TEXT数据列、同一数据表可以有更多个索引等等。

不同类型的数据表有着不同的索引特性，其中最为明显的差异是：根据所使用的MySQL版本，可能无法对某些类型的数据列进行索引。比如说，如果MySQL早于3.23版本，就只能使用ISAM数据表，这就意味着将无法对包含有NULL值的数据列进行索引。反过来讲，如果想让索引具备某些特定的属性，就不能把数据表创建为某种特定的类型。比如说，如果想索引一个BLOB数据列，就必须使用MyISAM或BDB数据表。

为了让索引具备某种特定的属性，可能需要把数据表从一种类型转换为另一种类型，这种类型转换工作可以用ALTER TABLE语句来完成。比如说，假设使用的是MySQL 3.23或以后的



版本，但数据表有些是较早期创建的ISAM类型，现在，如果想利用MyISAM所提供的高级索引功能，只需使用ALTER TABLE语句简单地把它们转换成MyISAM存储格式就行了：

```
ALTER TABLE tbl_name TYPE = MYISAM;
```

## 2. 索引的创建

MySQL能够创建以下几种类型的索引：

- 普通（非惟一化）索引。这能让你享受到索引带来的好处，但允许索引键值重复。
- 惟一化索引。这种索引不允许键值重复。建立在单个数据列上的惟一化索引要求该数据列里的数据值不得重复。建立在多个数据列上的（复合型）惟一化索引要求各有关数据行的不同组合不得出现重复的数据值。
- 供全文本检索操作使用的FULLTEXT索引。这种索引类型只能用在MyISAM数据表里。（详细情况请参见第3.9节的内容。）

既可以使用CREATE TABLE语句在创建一个数据表的同时创建索引，也可以使用CREATE INDEX或ALTER TABLE语句将索引添加到现有的数据表上去。CREATE INDEX语句最早出现于MySQL 3.22版本，在更早的MySQL版本里，需要使用ALTER TABLE语句。（MySQL在内部是把CREATE INDEX语句映射到ALTER TABLE操作上的。）

与CREATE INDEX语句相比，ALTER TABLE语句能够完成更多的工作。可以用它来创建普通索引、UNIQUE（惟一化）索引、PRIMARY KEY（主键）或是FULLTEXT（全文本）索引：

```
ALTER TABLE tbl_name ADD INDEX index_name (index_columns);
ALTER TABLE tbl_name ADD UNIQUE index_name (index_columns);
ALTER TABLE tbl_name ADD PRIMARY KEY (index_columns);
ALTER TABLE tbl_name ADD FULLTEXT (index_columns);
```

*tbl\_name*是要在其中添加索引的数据表的名字，*index\_columns*表明将使用哪些数据列来建立索引。如果想用多个数据列来建立索引，就要用逗号把各有关数据列的名字分隔开。索引名*index\_name*是一个可选项，如果没有给出这个名字，MySQL将根据第一个被索引数据列的名字去挑选一个索引名。ALTER TABLE允许用一条语句对数据表做出多项改变，所以完全可以同时创建出多个索引来。（这要比用一组语句逐个地添加各个索引的做法快很多。）

如果想让某个索引只包含惟一化的键值，可以将该索引创建为一个PRIMARY KEY 或者是一个UNIQUE索引。这两种索引有很多相似之处。事实上，可以把PRIMARY KEY看做是一个名为PRIMARY的UNIQUE索引。这两种索引的不同之处是：

- 每个数据表只能有一个PRIMARY KEY，因为不可能有两个名为PRIMARY的索引。但同一个数据表却允许有多个UNIQUE索引——尽管很少有必要这样做。
- PRIMARY KEY不允许包含NULL值，UNIQUE索引却可以。如果某个UNIQUE索引的确包含有NULL值，那它通常会包含多个NULL值。造成这种现象的原因是根本无法确定某个NULL值与另一个NULL值是否都代表着同样的数据值，所以不能认为它们是相等的。（注意：BDB数据表是一个例外——BDB数据表中的每个UNIQUE索引只允许有一个NULL值。）

CREATE INDEX语句能够给数据表添加一个普通索引、UNIQUE索引或FULLTEXT索引，

但不能用来添加PRIMARY KEY:

```
CREATE INDEX index_name ON tbl_name (index_columns);
CREATE UNIQUE INDEX index_name ON tbl_name (index_columns);
CREATE FULLTEXT INDEX index_name ON tbl_name (index_columns);
```

*tbl\_name*、*index\_name*以及*index\_columns*与它们在ALTER TABLE语句中的含义是一样的。与ALTER TABLE语句不同的是,CREATE INDEX语句中的索引名不是可选项,并且不允许用一条语句来创建多个索引。

通过CREATE TABLE语句为新建数据表创建索引的语法与ALTER TABLE语句的语法差不多,只是在有关数据列的定义里会多出一个索引创建子句:

```
CREATE TABLE tbl_name
(
    ... column declarations ...
    INDEX index_name (index_columns),
    UNIQUE index_name (index_columns),
    PRIMARY KEY (index_columns),
    FULLTEXT index_name (index_columns),
    ...
);
```

类似于ALTER TABLE语句,CREATE TABLE语句中的INDEX、UNIQUE、FULLTEXT子句中的索引名也是可选的。如果没有给索引起名字,MySQL就会自动挑选一个。

作为一种特例,可以通过把PRIMARY KEY关键字添加到数据列定义末尾的办法来创建一个基于单个数据列的PRIMARY KEY。从MySQL 3.23版本开始,也可以对UNIQUE索引做同样的事情。比如说,下面这个语句:

```
CREATE TABLE mytbl
(
    i INT NOT NULL PRIMARY KEY,
    j CHAR(10) NOT NULL UNIQUE
);
```

等价于下列语句:

```
CREATE TABLE mytbl
(
    i INT NOT NULL,
    j CHAR(10) NOT NULL,
    PRIMARY KEY (i),
    UNIQUE (j)
);
```

上面两个例子在创建数据表的时候都把被索引的数据列设定为NOT NULL。对于ISAM数据表(以及MySQL 4.0.2版本之前的HEAP数据表),必须这样做,因为这种数据表不允许索引建立在可能包含有NULL值的数据列上。对于其他类型的数据表,被索引的数据列允许包含NULL值,但建立在其上的索引就不能是PRIMARY KEY了。

如果想要索引（注意，“索引”在这里用做动词）某字符串数据列的前缀（即数据列值的前  $n$  个字节），该数据列在索引定义中的语法就是 `col_name(n)` 而不是简单的 `col_name`。比如说，下面的语句创建了一个有两个CHAR数据列的数据表，但用这两个数据列创建的索引却仅使用了它们的前10个字节：

```
CREATE TABLE mytbl
(
    name      CHAR(30) NOT NULL,
    address   CHAR(60) NOT NULL,
    INDEX (name(10),address(10))
);
```

ISAM、MyISAM、HEAP以及BDB数据表都支持索引前缀，但InnoDB数据表不支持。

前缀的长度（类似于数据列的长度概念）指的是字节而不是字符。对单字节字符集来说，字节长度与字符长度是一样的；对多字节字符集来说，它们可就不一样了。MySQL会把尽可能多的完整字符保存到索引值里去。比如说，假设索引前缀是5字节长、数据列值由双字节字符组成，那么，索引值将包含2个字符而不是2.5个字符。

在大多数场合，可以选择是索引某数据列的前缀还是索引该数据列的全部内容，但在某些场合，必须或者只能索引某数据列的前缀：

- 建立在BLOB或TEXT数据列上的索引（如果有关的数据表类型允许对这两种数据列进行索引的话）只能使用它们的前缀。这个前缀的最大长度是255字节。
- 索引行（index row）的长度等于组成该索引的各有关数据列的索引部分的长度总和。如果各有关数据列的索引部分的长度加起来超过了该索引行的最大长度，就需要通过索引数据列前缀来使这个索引变得“窄”一些。比如说，一个MyISAM数据表包含有两个CHAR(255)数据列c1和c2，想用这两个数据列创建一个索引。如果不做处理，索引行的长度将等于255+255，超出了MyISAM数据表的每个索引行最多只能有500个字节的上限。因此，如果想创建这个索引，就必须对这两个数据列中的一个或者两个进行前缀化索引才能达到目的。

索引某数据列的前缀会对今后数据列的修改带来一些限制。比如说，假设用某个数据列的前缀创建了一个索引，那么，如果想把该数据列的长度缩短到小于该索引前缀的长度，就必须先丢弃该索引，然后再重新定义该数据列（以及必要的索引）。例如，假如索引的是一个40字节长的CHAR数据列的前30个字节，但后来发现自己从未在这个数据列里存放过超过20个字节长的数据。因此，决定把这个数据列的长度修改为20个字节长以节省数据表里的一部分空间。为实现这一想法，必须先丢弃那个30个字节长的前缀型索引，然后缩短该数据列的长度，再然后才是重新添加必要的索引——新索引的长度是20个或更少的字节。

FULLTEXT索引中的数据列没有前缀，即使为FULLTEXT索引设定了一个前缀长度，它也会被忽略。

### 3. 索引的丢弃

可以用DROP INDEX 或ALTER TABLE语句来丢弃索引。类似于CREATE INDEX语句，DROP INDEX语句最早出现于MySQL 3.22版本，在MySQL内部是被当做ALTER TABLE语句来

处理的，并且不能用来影响PRIMARY KEY。索引丢弃语句的语法如下所示：

```
DROP INDEX index_name ON tbl_name;
ALTER TABLE tbl_name DROP INDEX index_name;
ALTER TABLE tbl_name DROP PRIMARY KEY;
```

前两个语句是等价的。第三个语句是用来丢弃PRIMARY INDEX的（它也只有这一个用途），因为每个数据表只能有一个PRIMARY KEY，所以这条语句不会产生二义性。如果数据表没有被明确地创建为PRIMARY KEY但有一个或多个UNIQUE索引，MySQL将丢弃该数据表的第一个UNIQUE索引。

如果从数据表里丢弃（删除）了一些数据列，与之有关的索引将受到影响。如果丢弃的数据列是某个索引的组成部分之一，该数据列将从该索引中删除掉（但该索引仍会存在）。如果构成某个索引的数据列全都被丢弃了，这个索引也将不复存在。

### 3.3.5 变更数据表的结构

ALTER TABLE语句在MySQL里有很多用途，可以用它来做许多事情。我们已经见过它的一些用法了（如本章中的改变数据表类型、创建和丢弃索引，第2章里的重新编排序列编号等等）。还可以用ALTER TABLE语句来重命名数据表、添加或丢弃数据列、改变数据列类型，等等。在这一小节中，将介绍这条语句的其他一些用法。ALTER TABLE语句的完整语法可以在附录D中查到。

当发现某个数据表的结构已不再能满足使用要求时，就该想到ALTER TABLE语句。可能需要用这个数据表来存放更多的信息，也可能是这个数据表里的某些信息已不再需要了。或许是现有的数据列宽度太窄，或许是当初把它们定义得过宽而现在想把它们缩短一些以节省空间并改善查询的性能，甚至有可能是在使用CREATE TABLE语句时敲错了数据表的名字。下面给出了一些例子：

- 你正在进行一个研究项目，使用了一个AUTO\_INCREMENT数据列来对研究记录进行编号。起初，研究经费只够生成最多50 000条研究记录，所以把该数据列的类型定义为SMALLINT UNSIGNED，即最多能够容纳65 535个彼此不同的编号值。但研究经费后来又增加到足够再生成50 000条研究记录，于是，你决定使用一个更大的类型来容纳更多的记录编号。
- 数据列的尺寸也许需要往另一个方向改变。比如说，当初创建了一个CHAR(255)数据列，但后来却发现数据表里根本没有长度超过100个字符的数据值。于是，你决定缩短数据列以节省空间。
- 你想把数据表转换为另外一种类型以利用那种类型所提供的功能。比如说，ISAM数据表不允许NULL值出现在被索引的数据列里，如果的确需要索引一个含有NULL值的数据列，就需要把它转换为MyISAM数据表。

下面是ALTER TABLE语句的语法：

```
ALTER TABLE tbl_name action, ... ;
```



每一个动作 (action) 都代表着一个想对数据表进行的修改。有些数据库引擎只允许每个 ALTER TABLE 语句完成一个动作, 但MySQL却允许用一条 ALTER TABLE 语句完成多个动作——只要用逗号把各个动作分隔开就行了。MySQL对 ALTER TABLE 语句的这一扩展是很有用的, 因为有些数据表修改操作是无法用只能完成一个动作的 ALTER TABLE 语句去完成的。比如说, 如果要把所有的 VARCHAR 数据列都改变为 CHAR 数据列, 使用多个每次只能完成一个动作的 ALTER TABLE 语句是不会成功的——必须同时改变它们。

下面这些示例给出了 ALTER TABLE 语句的一些功能:

- **重命名数据表。**使用 RENAME 子句给出新数据表的名字:

```
ALTER TABLE tbl_name RENAME TO new_tbl_name;
```

另外一种重命名数据表的方法是使用 RENAME TABLE 语句, MySQL 3.23.23 及以后的版本都提供有这条语句, 其具体语法如下所示:

```
RENAME TABLE old_name TO new_name;
```

有一件事情是 RENAME TABLE 语句可以完成但 ALTER TABLE 语句所不能完成的, 那就是 ALTER TABLE 语句允许在一个语句中同时对多个数据表重命名。比如说, 可以像如下所示这样交换两个数据表的名字:

```
RENAME TABLE t1 TO tmp, t2 TO t1, tmp TO t1;
```

如果在数据表名的前面加上一个数据库名修饰符 (即写成 *db\_name.tbl\_name* 的样子), 就可以用重命名操作把数据表从一个数据库移到另一个数据库里去。下面两条命令都能把数据表 *t* 从 *sampdb* 数据库移到 *test* 数据库里去:

```
ALTER TABLE sampdb.t RENAME TO test.t;
RENAME TABLE sampdb.t TO test.t;
```

注意, 不能把数据表重命名为一个已经存在的名字。

- **改变数据列的类型。**要想改变数据列的类型, 可以使用 CHANGE 或是 MODIFY 子句。假设 *mytbl* 数据表里有一个 SMALLINT UNSIGNED 数据列, 如果想把它改变为 MEDIUMINT UNSIGNED, 可以使用下列命令中的任意一个:

```
ALTER TABLE mytbl MODIFY i MEDIUMINT UNSIGNED;
ALTER TABLE mytbl CHANGE i i MEDIUMINT UNSIGNED;
```

为什么数据列的名字在 CHANGE 子句里出现了两次? 因为 CHANGE 子句可以完成但 MODIFY 子句不能完成的一件事情是: CHANGE 子句不仅可以改变数据列的类型, 还可以重命名数据列。如果想在改变数据列类型的同时将数据列 *i* 重新命名为 *j*, 可以像下面这样进行操作:

```
ALTER TABLE mytbl CHANGE i j MEDIUMINT UNSIGNED;
```

在使用 CHANGE 子句的时候, 有件事要特别注意: 在给出打算改变的数据列名之后, 必须完整地给出一个包括数据列名在内的数据列定义。数据列定义里的数据列名不能省略, 哪怕它与旧名字完全一样也要这么做。

MySQL 4.1 及以后的版本允许为每一个数据列分别指定一个不同的字符集，即允许在数据列定义里使用 CHARACTER SET 属性来改变它的字符集：

```
ALTER TABLE t MODIFY c CHAR(20) CHARACTER SET ucs2;
```

改变数据列类型的一个重要原因是为了提高对两个数据表的数据列进行比较的关联查询操作 (JOIN) 的效率。当两个数据列是同一种类型时，它们之间的比较操作要执行得快一些。假定正在运行一个如下所示的查询：

```
SELECT ... FROM t1, t2 WHERE t1.name = t2.name;
```

如果 t1.name 是 CHAR(10) 而 t2.name 是 CHAR(15)，那么，这个查询的执行速度将不如这两个数据列的类型都是 CHAR(15) 时快。可以使用下面两个命令中的任意一个来改变 t1.name 的类型，从而使它们成为同一种类型：

```
ALTER TABLE t1 MODIFY name CHAR(15);  
ALTER TABLE t1 CHANGE name name CHAR(15);
```

在 MySQL 3.23 之前的版本里，被关联的两个数据列最好是同一种类型。否则，因为无法利用索引来完成比较操作，关联查询将执行得非常慢。在 3.23 及以后的版本里，虽然两个不同数据列之间的比较操作也可以利用索引来完成，但如果它们是同一种类型，查询将执行得更快。

- 将数据表由可变长度数据行转变成固定长度数据行。假如有一个用下面这条语句创建的 chartbl 数据表：

```
CREATE TABLE chartbl (name VARCHAR(40), address VARCHAR(80));
```

这个数据表里的数据列都是 VARCHAR 类型，下面把它们转换成 CHAR 数据列来看看能得到哪些性能改进。（如果数据表使用的是 ISAM 或 MyISAM 存储格式，固定长度的数据行通常要比可变长度的数据行有更快的处理速度。）这里的问题是必须用同一条 ALTER TABLE 语句来同时改变所有的数据列。不能每次只改变一个数据列，因为这种尝试将毫无效果。（你不妨试验一下：只改变其中的某一个数据列，然后运行 DESCRIBE chartbl 命令，就会发现该数据列仍将被定义为 VARCHAR！）这个现象的原因是：如果每次只改变一个数据列，MySQL 就会因数据表里仍包含有可变长度的数据列而自动地把改变的那个数据列再改变回 VARCHAR 类型以节省空间。要想解决这个问题，就必须同时改变所有的 VARCHAR 数据列：

```
ALTER TABLE chartbl MODIFY name CHAR(40), MODIFY address CHAR(80);
```

现在，DESCRIBE 命令将显示数据表里包含的都是 CHAR 数据列了。正是因为这类操作的需要，才使得 MySQL 允许用同一条 ALTER TABLE 语句完成多个动作的做法成为一种非常有价值的功能扩展。

当打算转换数据表的类型时，有一点要特别注意：不存在与 BLOB 和 TEXT 这两种可变长度类型等价的固定长度类型。因此，只要数据表里有 BLOB 或 TEXT 数据列，把它转换为固定长度数据行格式的一切努力都将徒劳无功——因为只要数据表里有一个可变长度的数据列，就会使该数据表里的数据行都是可变长度的。

- 将数据表由固定长度数据行转变成可变长度数据行。假设发现数据表chartbl使用固定长度的数据行时确实要快一些，但它占用的存储空间却太多，所以决定把它转换回原来的存储格式以节省空间。这一方向上的数据表转换操作非常简单，只需把一个CHAR数据列转换为VARCHAR就足够了，MySQL将自动地把其他CHAR数据列都转换过来。仍以chartbl数据表为例，可以用下面的任意一条语句：

```
ALTER TABLE chartbl MODIFY name VARCHAR(40);
ALTER TABLE chartbl MODIFY address VARCHAR(80);
```

- 转换数据表类型。如果想把数据表从一种存储格式转换为另一种存储格式，请使用一个TYPE子句来改变数据表的类型：

```
ALTER TABLE tbl_name TYPE = tbl_type;
```

tbl\_type是类型名标识符，它的可取值包括ISAM、MYISAM、HEAP、BDB或INNODB（不区分字母的大小写）。

在把MySQL软件升级到一个较新的版本（新版本提供了更高级的数据表处理功能）之后，通常还需要改变各有关数据表的类型。比如说，即使把MySQL软件从3.23之前的版本升级到了3.23或者更高的版本，老数据表也仍都是ISAM格式。要想把它们都转换为MyISAM格式，就要对每一个数据表使用下面的语句：

```
ALTER TABLE tbl_name TYPE = MYISAM;
```

这样，就能享用MyISAM提供的、ISAM不具备的各种功能了，这些都已经在本章前面进行了讨论。举例来说，MyISAM数据表是与计算机硬件无关的，所以即便两台（甚至更多台）MySQL服务器主机有着不同的硬件体系结构，也可以通过直接拷贝数据表文件把它们从这台机器移动到那台机器里去。此外，MyISAM数据表还允许对BLOB和TEXT数据列进行索引，并允许被索引的数据列包含NULL值。

改变数据列类型的另一个重要原因是为了让它支持事务处理机制。假设有一个MyISAM数据表并发现有一个应用程序要使用这个数据表去进行一些事务操作（比如一旦执行出错则进行回滚）。MyISAM数据表不支持事务处理，但可以通过把它转换为BDB或InnoDB数据表使它支持事务处理机制：

```
ALTER TABLE tbl_name TYPE = BDB;
ALTER TABLE tbl_name TYPE = INNODB;
```

虽然ALTER TABLE语句有很多用途，但也有一些场合是不应该使用它的。下面给出了两个例子：

- HEAP数据表存在于内存里，当MySQL服务器关机时它们也就消失了。如果不希望自己的数据表在MySQL服务器关机时消失，就不应该把它们转换为HEAP类型。
- 如果打算用一组MyISAM数据表构成一个MERGE数据表，就应该避免使用ALTER TABLE语句去修改其中任何一个MyISAM数据表的结构——除非想对所有的MyISAM数据表以及MERGE数据表都做出同样的修改。要想让MERGE数据表正确地发挥作用，就必须让它与其子MyISAM数据表有着完全相同的结构。

### 3.4 获得关于数据库和数据表的信息

MySQL提供的SHOW语句有很多使用形式，可以用它们来查知关于数据库及其中的数据表的信息。SHOW语句能够随时反映数据库的内容并让你掌握数据表的结构，这对其他操作有很大的帮助。比如说，可以在使用ALTER TABLE语句之前先使用SHOW语句来了解各有关数据列的当前定义，这将有助于确定应该如何对数据列做出修改。

SHOW语句可以用来获取关于数据库和数据表的多方面信息：

- 列出服务器管理着的数据库：

```
SHOW DATABASES;
```

- 列出当前数据库或指定数据库里的数据表；

```
SHOW TABLES;
```

```
SHOW TABLES FROM db_name;
```

注意：SHOW TABLES语句并不显示TEMPORARY数据表。

- 显示关于数据表中的数据列或索引的信息：

```
SHOW COLUMNS FROM tbl_name;
```

```
SHOW INDEX FROM tbl_name;
```

DESCRIBE *tbl\_name*语句和EXPLAIN *tbl\_name*语句都是SHOW COLUMNS FROM *tbl\_name*语句的同义词。

- 显示关于当前数据库或者指定数据库中的数据表的描述信息：

```
SHOW TABLE STATUS;
```

```
SHOW TABLE STATUS FROM db_name;
```

这条语句是在MySQL 3.23.0版本中才开始出现的。

- 显示与数据表的当前结构相对应的CREATE TABLE语句：

```
SHOW CREATE TABLE tbl_name;
```

这条语句最早出现于MySQL 3.23.20版本。

SHOW语句的几种形式都有一个LIKE 'pat'子句，这个子句给出了一个用来限制输出内容范围的匹配模式。'pat'将被解释为一个SQL模式，这是一种以“%”和“\_”字符为通配符的匹配模式。比如说，下面这条语句将把当前数据库中以'geo'开头的数据表的名字列出来：

```
SHOW TABLES LIKE 'geo%';
```

出现在LIKE匹配模式里的通配符本身需要加上一个前导的反斜线字符（\）进行转义。这种情况多发生在需要对下划线字符“\_”进行匹配的场所，因为字符“\_”经常出现在数据库、数据表或数据列的名字当中。

mysqlshow命令能够提供很多与使用SHOW语句时获得的同样的信息，这个命令允许从shell（操作系统的命令解释器）获得关于数据库和数据表的信息：

- 列出服务器管理着的数据库：

```
% mysqlshow
```



- 列出指定数据库中的数据表:

```
% mysqlshow db_name
```

- 显示指定数据表中的数据列的信息:

```
% mysqlshow db_name tbl_name
```

- 显示关于指定数据表中的索引的信息:

```
% mysqlshow --keys db_name tbl_name
```

- 显示关于指定数据库中的数据表的描述信息:

```
% mysqlshow --status db_name
```

mysqldump 工具程序能够让你查看到 CREATE TABLE 语句对数据表结构的定义 (其作用与 SHOW CREATE TABLE 语句非常相似)。注意: 在用 mysqldump 工具程序查看数据表结构的时候, 一定要用 --no-data 选项来调用它, 以免破坏数据表里的数据!

```
% mysqldump --no-data db_name tbl_name
```

如果省略了数据表名, mysqldump 程序将依次显示数据库中所有数据表的结构。

在使用 mysqlshow 和 mysqldump 程序的时候, 还可以给出一些连接参数选项 (如 --host 或 --user 等)。

### 3.4.1 确定MySQL服务器所支持的数据表类型

在MySQL 3.23之前的版本里, ISAM是惟一可用的数据表类型。在MySQL 3.23及以后的版本里, MyISAM、MERGE以及HEAP都是可用的, 至于MySQL服务器是否还支持其他数据表类型的问题可以通过相应的SHOW VARIABLES语句查出来:

```
SHOW VARIABLES LIKE 'have_isam';
SHOW VARIABLES LIKE 'have_bdb';
SHOW VARIABLES LIKE 'have_inno%';
```

如果有关变量在查询结果中的取值是YES, 相应的数据表处理程序就是可用的; 如果是其他值或根本没有输出结果, 则说明没有相应的数据表处理程序。之所以要使用have\_inno%模式来判断InnoDB数据表处理程序的可用性, 是因为它能同时匹配have\_innodb和have\_innobase两个变量。(MySQL 3.23.30到3.23.36版本使用的是have\_innobase变量, 其他版本使用的是have\_innodb变量。)

可以根据数据表类型信息来判断MySQL服务器是否支持事务处理机制。我们知道, 支持事务处理机制的数据表类型有BDB和InnoDB两种, 因此, 只要用上面介绍的办法查知了它们的处理程序是否可用, 就可以知道MySQL服务器是否支持事务处理机制了。

从MySQL 4.1版本开始, 可以用SHOW TABLE TYPES语句直接得到一份数据表类型的清单:

```
mysql> SHOW TABLE TYPES;
```

Type	Support	Comment
MyISAM	DEFAULT	Default type from 3.23 with great performance
HEAP	YES	Hash based, stored in memory, useful for temporary tables
MERGE	YES	Collection of identical MyISAM tables
ISAM	YES	Obsolete table type; Is replaced by MyISAM
InnoDB	YES	Supports transactions, row-level locking and foreign keys
BDB	YES	Supports transactions and page-level locking

这里我们感兴趣的是输出列Support的取值：YES或NO分别表明相应的处理程序可用或不可用；DISABLED表明相应的处理程序可用但被禁用；DEFAULT表示是MySQL服务器默认使用的数据表类型。与DEFAULT相对应的处理程序应该是可用的。

### 3.4.2 检查数据表是否存在及其类型

在某些场合，需要在应用程序里检查某给定数据表是否存在，这可以用SHOW TABLES语句来查知：

```
SHOW TABLES LIKE 'tbl_name';
SHOW TABLES FROM db_name LIKE 'tbl_name';
```

如果SHOW语句列出了数据表的信息，则说明该数据表是存在的。用下面的任意一个语句也可以判断出数据表是否存在：

```
SELECT COUNT(*) FROM tbl_name;
SELECT * FROM tbl_name WHERE 0;
```

如果数据表存在，这两条语句都会成功返回，否则将执行失败。第一条语句更适用于MyISAM和ISAM数据表，因为不带WHERE子句的COUNT(\*)经过了高度的优化，执行起来非常快。（但这条语句在InnoDB或BDB数据表上的速度就没有这么快了，因为它需要进行全表扫描才能统计出数据行的总个数来。）第二条语句更具通用性，它在任何一种数据表类型上都执行得很快。这些查询非常适合用在以Perl或PHP语言编写出来的脚本程序里，因为可以对查询是否成功进行测试并根据测试结果来采取适当的动作。它们在需要用mysql程序来调用执行的批处理脚本里用处不大，因为一旦出现了错误，批处理脚本就会退出执行，根本没机会去进行出错处理（当然，也可以忽略该错误，但这就没有必要去运行这个查询了）。

至于判断某数据表是什么类型的问题，在MySQL 3.23.0及以后的版本里，可以使用SHOW TABLE STATUS语句；在MySQL 3.23.20及以后的版本里，可以使用SHOW CREATE TABLE语句。这两条语句的输出结果都能告诉你数据表到底是什么类型。MySQL 3.23.0之前的版本没有提供查看数据表类型的语句，但因为只有ISAM这一种数据表类型可用，所以根本用不着去检查某个数据表是什么类型。

### 3.5 涉及多个数据表的查询操作

把信息存放到数据库里的目的是为了在今后使用它们，如果不能利用信息来做一些事情，就没必要把它们存放到数据库里去。SELECT语句的用途就是从数据库里检索出想要的数据来。SELECT语句大概是SQL语言中使用最多的语句了，同时它也是最复杂的一个，用来选取数据行的筛选条件有些相当复杂，数据列之间的比较操作往往会涉及到多个数据表。

SELECT语句的基本语法如下所示：

SELECT selection_list	# 选择哪些数据列
FROM table_list	# 从哪些数据表选择数据行
WHERE primary_constraint	# 数据行必须满足哪些条件
GROUP BY grouping_columns	# 如何对结果进行归组
ORDER BY sorting_columns	# 如何对结果进行排序
HAVING secondary_constraint	# 数据行必须满足的次要条件
LIMIT count;	# 对结果个数的限制

在这个语法里，除单词“SELECT”和用来给出检索条件的selection\_list部分外，其余东西都是可选的。有些数据库还要求FROM子句不得省略，但MySQL没有这个要求，这使你能够方便地对表达式进行求值而无须用到任何数据表，如下所示：

```
SELECT SQRT(POW(3,2)+POW(4,2));
```

在第1章里，我们把大部分注意力都放在了单个数据表的SELECT语句上，集中讨论了如何选取数据列以及WHERE、GROUP BY、ORDER BY、HAVING和LIMIT子句。在本节里，将集中讨论SELECT语句最难掌握的一个方面——如何写出关联查询，即如何使用SELECT语句从多个数据表里检索出想要的信息。将讨论MySQL所支持的关联类型、它们的含义以及如何写出它们。这将帮助你更有效地用好MySQL，因为在许多情况下，正确地写出一个查询的关键其实就是怎样才能把各有关数据表按正确的顺序关联起来。

在使用SELECT语句的时候，经常会遇到这样的情况：对于第一次遇到的新问题，不容易一眼看出怎样书写SELECT查询才能正确地解决这个问题，但在成功地解决了这个问题之后，因为有了这次的经验，今后再遇到类似的问题时，就能够比较轻松地解决它们了。与数据库有关的问题可以说是千变万化，它们几乎都需要通过SELECT语句才能得到解决，而要想有效地通过SELECT语句来解决它们，经验是一个至关重要的因素。

在具备了足够的经验之后，就能更加轻松地运用关联查询来解决新的问题。比如说，在遇到一个新问题的时候，你可能会对自己说：“这好办，这事可以用左关联（LEFT JOIN）来解决”，或者是：“啊哈，用公共键对数据列做一次三方关联就行了。”（说实在的，我在这么说的时侯还是有点迟疑的。经验的确非常重要，但绝不能因为自己有着丰富的经验而放松理论学习。如果认为仅凭着经验就能包打天下，那可就大错特错了！）

为了演示关联检索操作在MySQL里的用法，本节准备了很多例子。大部分例子都要用到如下所示的两个数据表t1和t2（这两个数据表都不大，也都很简单，目的是为了让大家能够清楚地看到每种关联操作的执行效果）：

数据表t1:	数据表t2:
+-----+-----+	+-----+-----+
i1   c1	i2   c2
+-----+-----+	+-----+-----+
1   a	2   c
2   b	3   b
3   c	4   a
+-----+-----+	+-----+-----+

### 3.5.1 单关联

单关联 (trivial join) 是一种最简单的关联操作——查询只涉及一个数据表。也就是说，结果集里的数据行都是从同一个数据表里选取出来的：

```
mysql> SELECT * FROM t1;
```

+-----+-----+
i1   c1
+-----+-----+
1   a
2   b
3   c
+-----+-----+

有些人根本就不把这种形式的SELECT查询看做是一种关联操作，他们认为“关联检索至少要涉及两个或者更多个数据表”。在我看来，这只是看问题的角度不同而已。

### 3.5.2 全关联

当在SELECT语句的FROM子句里给出了多个数据表的名字（名字之间要用逗号隔开）时，MySQL就将进行一次全关联（full join）操作。比如说，下面这条语句将对数据表t1和t2进行全关联，即把数据表t1中的每一个数据行依次与数据表t2中的每一个数据行组合在一起：

```
mysql> SELECT t1.*, t2.* FROM t1, t2;
```

+-----+-----+-----+-----+
i1   c1   i2   c2
+-----+-----+-----+-----+
1   a   2   c
2   b   2   c
3   c   2   c
1   a   3   b
2   b   3   b
3   c   3   b
1   a   4   a
2   b   4   a
3   c   4   a
+-----+-----+-----+-----+

全关联也叫做“交叉关联”（cross join），因为每一个数据表中的每一个数据行将依次与其



他数据表中的每一个数据行进行“交叉”，其最终结果将是所有数据表里的所有数据行的全排列组合，即所谓的“笛卡儿乘积”。因为输出行的总数等于各有关数据表里的数据行个数的总乘积，所以极可能产生数量相当巨大的输出行。比如说，数据行个数分别是100个、200个、300个的三个数据表的全关联操作将产生 $100 \times 200 \times 300 = 600$ 万个输出行！虽然每一个数据表都不大，但它们之间的全关联操作所产生的数据行却实在是太多了。在这类场合，为了把结果集的规模限制在可控制的范围内，通常要用WHERE子句来筛选掉绝大多数的输出行。

如果所添加的WHERE子句使数据表全关联操作的输出结果是根据各有关数据列值之间的某种匹配关系而生成的，这个关联操作就叫做“相等关联”(equi-join)，因为其结果集里的输出行都是且仅是那些能够满足WHERE子句所给定的“相等”关系的组合结果：

```
mysql> SELECT t1.*, t2.* FROM t1, t2 WHERE t1.i1 = t2.i2;
```

i1	c1	i2	c2
2	b	2	c
3	c	3	b

关联类型JOIN和CROSS JOIN与关联操作符“,”(逗号)是等价的。比如说，下面这几条语句有着同样的效果：

```
SELECT t1.*, t2.* FROM t1, t2 WHERE t1.i1 = t2.i2;
SELECT t1.*, t2.* FROM t1 JOIN t2 WHERE t1.i1 = t2.i2;
SELECT t1.*, t2.* FROM t1 CROSS JOIN t2 WHERE t1.i1 = t2.i2;
```

一般说来，MySQL优化程序会按照它认为将使输出行生成速度最快的方式来确定各有关数据表在关联操作中的扫描次序。但优化程序做出的选择并不总是最优的，如果你认为优化程序没能做出最佳选择，可以用STRAIGHT\_JOIN关键字来改写它做出的选择。STRAIGHT\_JOIN关联与刚才介绍的交叉关联情况差不多，但将迫使各有关数据表按FROM子句所给出的顺序进行关联。

STRAIGHT\_JOIN允许出现在SELECT语句中的两个地方：1) 可以把它放在SELECT关键字之后、selection\_list部分之前，这将使这条SELECT语句中所有的交叉关联操作都按照STRAIGHT\_JOIN方式进行；2) 可以把它放在FROM子句里。下面两个语句是等价的：

```
SELECT STRAIGHT_JOIN ... FROM t1, t2, t3 ... ;
SELECT ... FROM t1 STRAIGHT_JOIN t2 STRAIGHT_JOIN t3 ... ;
```

### 对数据列的名字进行“限定”

出现在SELECT语句里的每一个数据列的名字都必须毫无二义地对应于FROM子句中给出的某个数据表。如果FROM子句只给出了一个数据表，则不存在二义性问题，因为所有的数据列肯定都来自那个数据表。如果FROM子句列出了好几个数据表，那么，各数据表所独有的数据列的名字不会引起二义性问题；如果不同的数据表有同名的数据列，就必须按tbl\_name.col\_name语法给同名数据列加上一个数据表名作为限定符，即

明确地表明这个数据列到底属于哪一个数据表。我们来看一个例子：假设数据表mytbl1包含着数据列a和b，mytbl2包含着数据列b和c。那么，数据列a和c是不会引起二义性问题的，但数据列b就必须写成mytbl1.b或mytbl2.b才能保证它不会产生二义性问题：

```
SELECT a, mytbl1.b, mytbl2.b, c FROM mytbl1, mytbl2 ... ;
```

有时候，即使加上了数据表名作为限定符也仍不足以解决数据列名的二义性问题。比如说，如果某个查询命令需要把某个数据表关联到它自身并在查询命令里多次用到这个数据表，用这个数据表的名字来限定各有关数据列的名字的做法就行不通了。此时，需要用数据表的别名来实现你的意图：视具体情况给这个数据表起一个或者几个别名，再把各有关数据列的名字写成alias\_name.col\_name的形式。请看下面这个查询，它把数据表mytbl与它自身关联了起来。为了解决数据列名的二义性问题，在查询命令里给mytbl数据表起了一个别名m：

```
SELECT mytbl.col1, m.col2 FROM mytbl, mytbl AS m WHERE mytbl.col1 > m.col1;
```

### 3.5.3 左关联和右关联

相等关联只能检索出两个数据表中那些满足某种匹配关系的数据行。左关联和右关联不仅能够完成同样的工作，还能把其中某个数据表里的、在另一个数据表里没有匹配物的数据行也检索出来。在这一小节里，将以LEFT JOIN（左关联）为例进行讨论，它能把左数据表里的、在右数据表里没有匹配物的数据行也检索出来。RIGHT JOIN与LEFT JOIN恰好相反，它能把右数据表里的、在左数据表里没有匹配物的数据行也检索出来。（RIGHT JOIN最早出现于MySQL 3.23.25版本。）

LEFT JOIN的工作情况如下所述：首先，要指定一个或者多个用来匹配两个数据表里的数据行的数据列。当左数据表里的某个数据行与右数据表里的某个数据行相匹配时，这两个数据行的内容就将被选取为一个输出行。当左数据表里的某个数据行在右数据表里没有找到任何匹配时，它也将被选取为一个输出行，但这时它将与右数据表里的一个“伪造”数据行（这个“伪造”数据行的所有数据列都将被设置为NULL）进行组合。换句话说，不论能否在右数据表里找到匹配，左数据表里的每一个数据行都会在LEFT JOIN操作的结果集里有一个与之对应的输出行；如果左数据表里的某个数据行没能在右数据表里找到匹配，那么，在与之对应的输出行里，来自右数据表的所有数据列都将被设置为NULL值。

再看一次前面提到的两个数据表t1和t2：

数据表t1:	数据表t2:
<pre> +-----+   i1   c1   +-----+   1   a     2   b     3   c   +-----+ </pre>	<pre> +-----+   i2   c2   +-----+   2   c     3   b     4   a   +-----+ </pre>

如果用匹配关系t1.i1和t2.i2来交叉关联这两个数据表，只能得到两个分别对应于数据值2和3的输出行，因为只有这两个值是同时出现在两个数据表里的：

```
mysql> SELECT t1.*, t2.* FROM t1, t2 WHERE t1.i1 = t2.i2;
```

i1	c1	i2	c2
2	b	2	c
3	c	3	b

左关联将为数据表t1里的每一个数据行生成一个输出行，而不管数据表t2是否与之匹配。左关联的写法是：把两个数据表的名字写在LEFT JOIN（而不是逗号）两端——t1要写在左边，再用一个ON子句（不是WHERE子句）来给出匹配条件：

```
mysql> SELECT t1.*, t2.* FROM t1 LEFT JOIN t2 ON t1.i1 = t2.i2;
```

i1	c1	i2	c2
1	a	NULL	NULL
2	b	2	c
3	c	3	b

现在，多了一个与数据值1相对应的输出行——虽然它在数据表t2里没有匹配物。

LEFT JOIN特别适合用来找出左数据表里的、在右数据表里没有匹配物的数据行。其具体做法是：增加一条用来找出右数据表里的那个“伪造”数据行（即数据列全部取值为NULL的那个数据行）的WHERE子句：

```
mysql> SELECT t1.*, t2.* FROM t1 LEFT JOIN t2 ON t1.i1 = t2.i2
-> WHERE t2.i2 IS NULL;
```

i1	c1	i2	c2
1	a	NULL	NULL

一般说来，在使用左关联的时候，真正关心的是左数据表里有哪些数据行在右数据表里没有匹配。那些来自右数据表的NULL数据列根本没有显示的必要（因为早就知道它们全都是NULL了），所以通常根本用不着在SELECT语句的selection\_list部分把它们列举出来：

```
mysql> SELECT t1.* FROM t1 LEFT JOIN t2 ON t1.i1 = t2.i2
-> WHERE t2.i2 IS NULL;
```

i1	c1
1	a

LEFT JOIN实际上允许以两种方式给出匹配条件。ON子句是其中之一，无论打算关联的数据列名字是否相同，都可以使用ON子句：

```
SELECT t1.*, t2.* FROM t1 LEFT JOIN t2 ON t1.i1 = t2.i2;
```

用来给出匹配条件的另一种语法是使用USING ( )子句，它与ON子句在概念上是一样的，但要求被关联的数据列在两个数据表中必须有同样的名字。比如说，下面这个查询语句将使用数据列mytbl1.b和mytbl2.b进行关联：

```
SELECT mytbl1.*, mytbl2.* FROM mytbl1 LEFT JOIN mytbl2 USING (b);
```

LEFT JOIN还有几个同义词和变体。LEFT OUTER JOIN是LEFT JOIN的一个同义词。LEFT JOIN还有一种ODBC风格的表示法，这种表示法也能在MySQL下使用，如下所示（OJ的意思是“外关联”（outer join））：

```
{ OJ tbl_name1 LEFT OUTER JOIN tbl_name2 ON join_expr }
```

NATURAL LEFT JOIN与LEFT JOIN功能类似，它将以匹配左数据表和右数据表中所有同名数据列的方式来完成LEFT JOIN操作。

在使用LEFT JOIN的时候，有件事要特别注意：如果用来进行关联操作的数据列没有被定义为NOT NULL，所得到的结果集里就可能会多出一些不是由左关联操作生成的输出行来。这是因为，如果右数据表里用来进行关联操作的数据列取值为NULL，将无法把这个NULL值与用来标识无匹配输出行的“伪造”NULL值区分开来。

前面说过，LEFT JOIN在需要回答“哪些数据值是A数据表里有、而B数据表里没有的”这个问题时非常有用——只要对A、B两个数据表进行一次左关联查询，再把来自B数据表的数据列取值全都是NULL的输出行挑出来就行了。下面就是一个这类问题的例子，它比刚才那几个使用数据表t1和t2的例子要复杂得多。

在本书第1章提到的考试记分项目里，用student数据表来保存学生们的姓名，用event数据表来保存已发生的考试事件，用score数据表来保存每位学生在每次考试中的成绩。如果某位同学在考试当天生病了，score数据表里将没有这位同学在这次考试事件中的成绩——他需要参加补考。我们的问题是：怎样才能把缺勤考试的同学都查找出来以便通知他们参加补考呢？

这个问题其实就是把每一次考试事件中没有成绩记录的学生都查出来。它的另一种说法是怎样才能把在score数据表里没有成绩数据的学生和考试事件的组合情况都查出来。这种“在某个数据表里有、在另一个数据表里没有”的说法使我们立刻想到了LEFT JOIN。但这个关联操作不像前面几个例子那样简单，因为它要找的不是在某一个数据列里不存在的东西，而是要把两个数据列的一个组合给找出来。我们需要的是“学生/事件”的全排列组合，这些组合可以通过对student数据表和event数据表进行交叉关联的办法来生成，即：

```
FROM student, event
```

然后，要把这个交叉关联的结果与score数据表按如下所示的匹配条件进行一次LEFT JOIN关联：

```
FROM student, event
LEFT JOIN score ON student.student_id = score.student_id
AND event.event_id = score.event_id
```



注意，ON子句给出的匹配条件将使score数据表里的数据行根据它们与两个数据表（数据表student和event）的匹配情况来进行关联，这是解决这一问题的关键。因为LEFT JOIN的缘故，数据表student和event交叉关联出来的每一种组合（哪怕score数据表没有与之对应的考试分数记录）都将生成一个输出行。在这个结果集里，没有考试分数的输出行有着明显的标志，来自score数据表的数据列全都取值为NULL，可以用WHERE子句把这些记录找出来。因为来自score数据表的数据列全都取值为NULL，所以在WHERE子句里使用来自score数据表的哪一个数据列都能达到目的，但因为这里是要找出没有考试分数的记录，所以对score数据列进行测试的做法在概念上最明确：

```
WHERE score.score IS NULL
```

还可以用ORDER BY子句对最终的查询结果进行排序。这里又有两种最符合逻辑的排序方法，一种是按每位学生对各有关考试事件进行排序，另一种是按每次考试事件对各有关学生进行排序。这里采用的是第一种排序方法：

```
ORDER BY student.student_id, event.event_id
```

现在，只要把想在输出报告里看到的数据列的名字在这条SELECT语句的selection\_list部分全都列举出来就大功告成了。下面就是最终的查询命令：

```
SELECT
    student.name, student.student_id,
    event.date, event.event_id, event.type
FROM
    student, event
LEFT JOIN score ON student.student_id = score.student_id
                AND event.event_id = score.event_id
WHERE
    score.score IS NULL
ORDER BY
    student.student_id, event.event_id;
```

这个查询将产生如下所示的输出结果：

name	student_id	date	event_id	type
Megan	1	2002-09-16	4	Q
Joseph	2	2002-09-03	1	Q
Katie	4	2002-09-23	5	Q
Devri	13	2002-09-03	1	Q
Devri	13	2002-10-01	6	T
Will	17	2002-09-16	4	Q
Avery	20	2002-09-06	2	Q
Gregory	23	2002-10-01	6	T
Sarah	24	2002-09-23	5	Q
Carter	27	2002-09-16	4	Q
Carter	27	2002-09-23	5	Q
Gabrielle	29	2002-09-16	4	Q
Grace	30	2002-09-23	5	Q

这里有一个需要注意的细节：这份输出报告需要显示学生ID和事件ID。因为数据表student和score里都有student\_id数据列，所以你可能会认为在这条SELECT语句的selection\_list部分选取student.student\_id或score.student\_id是一样的。但事实却并非如此，因为解决这个问题的关键完全在于输出行中来自score数据表的字段全都取值为NULL，所以，如果在这条SELECT语句里选取的是score.student\_id数据列，它在输出报告里就会被显示为一个全都是NULL值的输出列。这一结论同样适用于在这条SELECT语句里选取event\_id数据列时的情况：数据表event和score里都有event\_id数据列，但因为score.event\_id数据列在输出报告里的取值永远是NULL，所以必须在这条SELECT语句里选取event.event\_id数据列。

### 3.5.4 使用子选择

对子选择的支持是从MySQL 4.1版本才开始出现的特色功能之一，这是一种人们期待已久的功能，它允许把一个SELECT查询嵌套在另一个SELECT查询当中。请看下面这个例子。考试记分项目把考试事件分为“考试”（'T'）和“测验”（'Q'）两种情况，下面这个查询将先查出与考试事件（'T'）相对应的事件ID，再利用它们去选取学生们在那些考试中的分数：

```
SELECT * FROM score
WHERE event_id IN (SELECT event_id FROM event WHERE type = 'T');
```

在很多时候，子选择可以被改写为关联，其具体做法将稍后介绍。如果MySQL是4.1之前的某个版本的话，就能体会到子选择改写技巧的重要性了。

与子选择有关的另一项MySQL新功能是根据另一个数据表的内容来删除或修改某个数据表中的记录。比如说，可能需要从某个数据表里把在另一个数据表里没有匹配的记录全都删除掉，或者需要从某个数据表里把一个或者多个数据列里的数据值拷贝到另一个数据表里去。这类操作将在第3.6节里进行讨论。

子选择有很多种写法和用法，本小节将只介绍其中的几种：

- **用子选择来生成一个参考值。**在这种情况下，需要用内层的SELECT语句来检索出一个数据值，然后把这个数据值用在外层SELECT语句的比较操作中。比如说，如果想查出学生们在2002年9月23日那一天的测验（'Q'）成绩，就需要使用一个内层SELECT语句来确定该次测验的事件ID，再在外层SELECT语句里用它去匹配学生们的分数记录：

```
SELECT * FROM score
WHERE event_id =
(SELECT event_id FROM event WHERE date = '2002-09-23' AND type = 'Q');
```

在使用这种形式（即内层查询的前面有一个比较操作符）的子选择时，内层关联不得生成多个输出值（即只能生成一个数据行或一个数据列）。如果生成了多个输出值，整个查询将以失败告终。（在某些场合，可能需要通过使用LIMIT 1把内层查询的结果限制为一个的办法来满足这一要求。）

这种形式的子选择很适合用来替代WHERE子句里的统计类函数。比如说，如果想知道哪位美国总统出生得最早，可能会使用如下所示的查询：

```
SELECT * FROM president WHERE birth = MIN(birth);
```

可惜，这个查询是无法工作的，因为MySQL不允许在WHERE子句里使用统计类函数。（WHERE子句是用来判断需要选取哪些记录的，而MIN()函数的返回值却必须在有关记录全都被选取出来之后才能确定下来。）此时，可以使用一个子选择来查出最早的出生日期，如下所示：

```
SELECT * FROM president
WHERE birth = (SELECT MIN(birth) FROM president);
```

- **EXISTS和NOT EXISTS子选择。**这两种形式的子选择将把外层查询检索到的数据值传递给内部查询，看它们是否满足内层查询所给出的匹配条件。注意，如果数据列的名字有可能产生二义性问题（即多个数据表有一些名字相同的数据列），别忘了用数据表的名字对它们做出限定（即把它们写成tbl\_name.col\_name的形式）。EXISTS和NOT EXISTS子选择非常适合用来检索某个数据表在另一个数据表里有或者没有匹配的记录。

仍以前面使用的数据表t1和t2为例：

数据表t1:		数据表t2:	
i1	c1	i2	c2
1	a	2	c
2	b	3	b
3	c	4	a

下面这个查询将把满足给定匹配条件（即两个数据表里都存在）的数据项找出来：

```
mysql> SELECT i1 FROM t1
-> WHERE EXISTS (SELECT * FROM t2 WHERE t1.i1 = t2.i2);
```

i1
2
3

NOT EXISTS将把不满足给定匹配条件（即一个数据表中存在，但另一个数据表里不存在）的数据项找出来：

```
mysql> SELECT i1 FROM t1
-> WHERE NOT EXISTS (SELECT * FROM t2 WHERE t1.i1 = t2.i2);
```

i1
1

在这两种形式的子选择里，内层查询中的星号(\*)代表的是外层查询的输出列名单。内层查询不必明确地列出各有关数据列的名字，因为内层查询是根据它是否会返回一些数据

行而不是根据它返回的数据行里是否包含有特定的数据值而被求值为真或者假的。MySQL允许在SELECT语句的`selection_list`部分自由地写出任何东西，但如果想明确地表明希望内层查询在成功时返回一个真值，就应该把它写成如下所示的样子：

```
SELECT i1 FROM t1
WHERE EXISTS (SELECT 1 FROM t2 WHERE t1.i1 = t2.i2);
SELECT i1 FROM t1
WHERE NOT EXISTS (SELECT 1 FROM t2 WHERE t1.i1 = t2.i2);
```

- **IN和NOT IN子选择。**在IN和NOT IN形式的子选择里，内层SELECT语句应该返回且只返回一个数据列，这个数据列里的值将由外层SELECT语句中的比较操作进行求值。比如说，刚才那两个EXISTS和NOT EXISTS查询可以用IN和NOT IN语法改写为如下所示的样子：

```
mysql> SELECT i1 FROM t1 WHERE i1 IN (SELECT i2 FROM t2);
+-----+
| i1 |
+-----+
| 2 |
| 3 |
+-----+
mysql> SELECT i1 FROM t1 WHERE i1 NOT IN (SELECT i2 FROM t2);
+-----+
| i1 |
+-----+
| 1 |
+-----+
```

把子选择查询改写为关联查询

在MySQL 4.1之前的版本里，子选择机制是不存在的。不过，子选择查询大都能被改写为关联查询。事实上，即便MySQL是4.1或更高的版本，尝试把子选择查询改写为关联查询也不是一个坏主意。与子选择查询相比，关联查询的执行效率往往要更高一些。

#### (1) 匹配型子选择查询的改写

下面是一个使用了子选择机制的查询示例，它将从score数据表里把学生们在考试事件('T')中的成绩（不包括学生们的测验成绩）都查出来：

```
SELECT * FROM score
WHERE event_id IN (SELECT event_id FROM event WHERE type = 'T');
```

这个子选择查询可以被改写为一个简单的关联查询：

```
SELECT score.* FROM score, event
WHERE score.event_id = event.event_id AND event.type = 'T';
```

我们再看一个例子。下面这个查询将把女学生们的成绩查出来：

```
SELECT * from score
WHERE student_id IN (SELECT student_id FROM student WHERE sex = 'F');
```

可以把它转换成一个如下所示的关联查询：

```
SELECT score.* FROM score, student
WHERE score.student_id = student.student_id AND student.sex = 'F';
```



把匹配型子选择查询改写为一个关联查询是有规律可循的。下面这种形式的子选择查询：

```
SELECT * FROM table1
WHERE column1 IN (SELECT column2a FROM table2 WHERE column2b = value);
```

可以转换为一个如下所示的关联查询：

```
SELECT table1.* FROM table1, table2
WHERE table1.column1 = table2.column2a AND table2.column2b = value;
```

## (2) 非匹配（即缺失）型子选择查询的改写

子选择查询的另一种常见用途是查找在某个数据表里有、但在另一个数据表里却没有的东西。正如前面看到的那样，这种“在某个数据表里有、在另一个数据表里没有”的说法通常都暗示着可以用一个LEFT JOIN来解决这个问题。请看下面这个子选择查询，它可以把没有出现在absence数据表里的学生（也就是那些从未缺过勤的学生）给查出来：

```
SELECT * FROM student
WHERE student_id NOT IN (SELECT student_id FROM absence);
```

这个子选择查询可以改写为如下所示的LEFT JOIN查询：

```
SELECT student.*
FROM student LEFT JOIN absence ON student.student_id = absence.student_id
WHERE absence.student_id IS NULL;
```

把非匹配型子选择查询改写为关联查询是有规律可循的。下面这种形式的子选择查询：

```
SELECT * FROM table1
WHERE column1 NOT IN (SELECT column2 FROM table2);
```

可以转换为一个如下所示的关联查询：

```
SELECT table1.*
FROM table1 LEFT JOIN table2 ON table1.column1 = table2.column2
WHERE table2.column2 IS NULL;
```

注意：这种改写要求数据列table2.column2声明为NOT NULL。

### 3.5.5 涉及多个数据表的UNION查询

如果想通过依次从多个数据表选取记录的办法来创建一个结果集，可以使用UNION语句。UNION语句最早出现于MySQL 4版本，在此之前，可以用其他办法（稍后将会介绍）来完成类似的工作。

在以下的例子里，将用到如下所示的三个数据表t1、t2和t3：

```
mysql> SELECT * FROM t1;
+-----+-----+
| i     | c     |
+-----+-----+
| 1     | red   |
| 2     | blue  |
| 3     | green |
+-----+-----+
```

```
mysql> SELECT * FROM t2;
```

```
+-----+-----+
| i      | c      |
+-----+-----+
|      -1 | tan    |
|       1 | red    |
+-----+-----+
```

```
mysql> SELECT * FROM t3;
```

```
+-----+-----+
| d          | i      |
+-----+-----+
| 1904-01-01 | 100    |
| 2004-01-01 | 200    |
| 2004-01-01 | 200    |
+-----+-----+
```

数据表t1和t2包含有整数和字符数据列，数据表t3包含有日期和整数数据列。要想写出一条组合多个检索操作的UNION语句，只需写出几个SELECT语句，再把关键字UNION放在它们中间就可以了。比如说，下面这条语句将把各数据表中的整数数据列都选取到同一个结果集里去：

```
mysql> SELECT i FROM t1 UNION SELECT i FROM t2 UNION SELECT i FROM t3;
```

```
+-----+
| i      |
+-----+
|      1 |
|      2 |
|      3 |
|     -1 |
|     100 |
|     200 |
+-----+
```

UNION语句具有以下特点：

- 在UNION查询的结果集里，输出列的名字和数据类型将由UNION查询中的第一个SELECT所选取的数据列的名字和类型来决定。UNION查询中的第二个以及其后的SELECT所选取的数据列的个数必须与第一个SELECT相同，但它们的名字和类型却不必相同。输出列是按位置（而不是名字）排列的，所以下面两个查询将返回不同的结果：

```
mysql> SELECT i, c FROM t1 UNION SELECT i, d FROM t3;
```

```
+-----+-----+
| i      | c      |
+-----+-----+
|      1 | red    |
|      2 | blue   |
|      3 | green  |
|     100 | 1904-01-01 |
|     200 | 2004-01-01 |
+-----+-----+
```

```
mysql> SELECT i, c FROM t1 UNION SELECT d, i FROM t3;
```

i	c
1	red
2	blue
3	green
1904	100
2004	200

在这两个例子里，选自数据表t1的数据列（i和c）决定了UNION结果中使用的类型。因为数据列i和c分别是整数和字符串类型，所以从数据表t3选取出来的数据值将发生类型转换。在第一个查询里，数据列d由日期转换为字符串，但碰巧没有丢失信息。在第二个查询里，数据列d由日期转换为整数（确实丢失了部分信息），数据列i由整数转换为字符串。

- 在默认情况下，UNION会把结果集里的重复项去掉，如下所示：

```
mysql> SELECT * FROM t1 UNION SELECT * FROM t2 UNION SELECT * FROM t3;
```

i	c
1	red
2	blue
3	green
-1	tan
1904	100
2004	200

数据表t1和t2都有一个“1, 'red'”数据行，但输出结果中只有一个这样的数据行。类似地，数据表t3有两个“'2004-01-01', 200”数据行，其中之一也会被去掉。

如果想保留重复项，请在第一个UNION关键字的后面加上关键字ALL：

```
mysql> SELECT * FROM t1 UNION ALL SELECT * FROM t2 UNION SELECT * FROM t3;
```

i	c
1	red
2	blue
3	green
-1	tan
1	red
1904	100
2004	200
2004	200

- 如果想对UNION结果进行排序，可以在最后一个SELECT语句的后面加上一个ORDER

BY子句，这个ORDER BY子句将作用于UNION查询的最终结果。注意：因为UNION查询的输出列使用的是第一个SELECT所选取的数据列的名字，所以在这种ORDER BY子句里就必须使用那些名字，而不能使用最后一个SELECT所选取的数据列的名字，在这两列数据列的名字不一样时更要注意这一点。

```
mysql> SELECT i, c FROM t1 UNION SELECT i, d FROM t3
-> ORDER BY c;
```

i	c
100	1904-01-01
200	2004-01-01
2	blue
3	green
1	red

UNION查询中的各个SELECT语句也允许有它自己的ORDER BY子句。在这样做的时候，必须把SELECT语句（包括它的ORDER BY子句）放在括号里：

```
mysql> (SELECT i, c FROM t1 ORDER BY i DESC)
-> UNION (SELECT i, c FROM t2 ORDER BY i);
```

i	c
3	green
2	blue
1	red
-1	tan

- LIMIT也可以用在UNION查询中，其具体用法与ORDER BY差不多。如果是被添加到整个语句的末尾，它就将对整个UNION查询产生作用：

```
mysql> SELECT * FROM t1 UNION SELECT * FROM t2 UNION SELECT * FROM t3
-> LIMIT 1;
```

i	c
1	red

如果被当做某个SELECT语句的一部分而被放在了括号里，它将只对那条SELECT语句产生作用：

```
mysql> (SELECT * FROM t1 LIMIT 1)
-> UNION (SELECT * FROM t2 LIMIT 1)
-> UNION (SELECT * FROM t3 LIMIT 1);
```

+-----+-----+	
i	c
+-----+-----+	
1	red
-1	tan
1904	100
+-----+-----+	

- UNION查询并非只能对多个不同的数据表进行选取。可以利用不同的筛选条件去选取同一个数据表的不同子集。在某些场合，用UNION查询来替代一组对同一个数据表进行的SELECT查询其效果可能更好，因为所有的数据行都将出现在同一个结果集而不是分散在好几个结果集里。

在MySQL 4之前，UNION查询机制是不存在的，但可以采取其他办法来实现类似的功能。其具体做法是：先把从各有关数据表里选取出来的数据行都存放到一个临时数据表里，最后再对这个数据表的内容进行检索。这在MySQL 3.23及以后的版本里是很容易做到的——可以让MySQL服务器去创建这个用来存放中间结果的数据表。更简便的做法是把这个数据表创建一个TEMPORARY数据表，这样，当与MySQL服务器的本次会话结束时，系统就会自动地丢弃它。要是你很注重性能，还可以把它创建一个HEAP数据表（这种数据表将一直驻留在内存里）。

```
CREATE TEMPORARY TABLE tmp TYPE = HEAP SELECT ... FROM t1 WHERE ... ;
INSERT INTO tmp SELECT ... FROM t2 WHERE ... ;
INSERT INTO tmp SELECT ... FROM t3 WHERE ... ;
...
SELECT * FROM tmp ORDER BY ... ;
```

上面这段代码里的tmp数据表是一个TEMPORARY数据表，所以当客户程序结束本次会话的时候，MySQL服务器将自动地丢弃它。（当然，也可以在用完这个tmp数据表之后亲自去丢弃它，以便让MySQL服务器能够在第一时间释放它所占用的资源。这是一种很好的习惯，当还要继续进行其他的查询（特别是还要用到其他一些HEAP数据表）的时候，这种做法将大大减轻MySQL服务器的负担。）

那些早于3.23的MySQL版本也可以用类似的思路来实现UNION查询机制，但在细节方面有所不同，因为MySQL的早期版本根本没有HEAP数据表类型、TEMPORARY数据表和CREATE TABLE ... SELECT语句等。要想采用刚才的做法，就得先明确地创建一个供临时使用的数据表（注意：因为早期的MySQL版本只有ISAM这一种数据表类型可用，所以无法使用TYPE选项），然后再把各有关数据行检索到其中去。注意：在用完这个临时数据表之后，必须明确地发出一条DROP TABLE语句去丢弃它，因为MySQL服务器不会自动丢弃它。

```
CREATE TABLE tmp (column1, column2, ...);
INSERT INTO tmp SELECT ... FROM t1 WHERE ... ;
INSERT INTO tmp SELECT ... FROM t2 WHERE ... ;
INSERT INTO tmp SELECT ... FROM t3 WHERE ... ;
SELECT * FROM tmp ORDER BY ... ;
DROP TABLE tmp;
```



在无法使用UNION查询的情况下，如果需要在一组结构完全相同的MyISAM数据表上运行一个UNION类型的查询，还可以考虑先建立一个MERGE数据表、再对这个MERGE数据表进行查询的方案。（事实上，即使能够进行UNION查询，这一方案也很值得考虑——因为MERGE数据表上的查询通常要比相应的UNION查询简单一些。）MERGE数据表上的查询与对构成MERGE数据表的各个数据表的各有关数据列进行UNION查询的效果是一样的。简单地说，MERGE数据表上的SELECT查询相当于一个UNION ALL查询（不剔除结果集里的重复项），而MERGE数据表上的SELECT DISTINCT查询则相当于一个UNION查询（会剔除结果集里的重复项）。

### 3.6 涉及多个数据表的删除和修改操作

在MySQL 4之前的版本里，每条DELETE语句只能对一个数据表进行操作，而且不得引用其他数据表里的数据列。可有些场合却需要能够根据另一个数据表里是否有与之匹配的记录而去删除某个数据表里的记录，MySQL从4.0.0版本开始增加了这项功能。类似地，有些场合又需要能够用另一个数据表的记录去修改某个数据表里的记录，MySQL从4.0.2版本开始增加了这项功能。本节将对涉及多个数据表的DELETE和UPDATE操作进行讨论。这类语句与关联查询有着密切的关系，所以大家一定要熟练掌握前面的内容。

对于只涉及单个数据表的DELETE或UPDATE操作，所用到的数据列都来自同一个数据表，所以不必用数据表的名字去限定各有关数据列的名字。比如说，如果想从数据表t里把id值大于100的所有记录都删除掉，只需写出下面这样的语句就行了：

```
DELETE FROM t WHERE id > 100;
```

可是，如果删除记录的依据不是这些记录本身所具有的某种属性而是它们与另一个数据表里的有关记录的某种关系时，又该怎么办呢？比如说，如果需要从数据表t里把那些其id值也能在数据表t2里找到的记录都删除掉，该怎么办呢？

涉及多个数据表的DELETE语句并不难写：把所涉及的数据表列举在FROM子句里、再把用来找出待删除记录的匹配条件写在WHERE子句里就行了。比如说，下面这条语句将从数据表t1里把那些其id值在数据表t2里有匹配的记录全都删除掉：

```
DELETE t1 FROM t1, t2 WHERE t1.id = t2.id;
```

注意，这个DELETE操作所涉及的数据表的名字全都列举在了FROM子句里，这与写关联查询时的做法是一样的。此外，如果某个数据列名出现在了多个数据表里，还必须用数据表名对它进行限定以避免出现二义性问题，这与写关联查询时的做法也是一样的。

涉及多个数据表的DELETE语句还允许用同一条语句从多个数据表里同时把有关的记录都删除掉。比如说，如果想同时从数据表t1和t2里把那些id值相互匹配的记录都删除掉，就需要把这两个数据表的名字都写在DELETE关键字的后面：

```
DELETE t1, t2 FROM t1, t2 WHERE t1.id = t2.id;
```

如果需要删除的是无匹配的记录，又该怎么办呢？好办，沿用编写用来查找无匹配记录的SELECT语句时的做法（也就是使用LEFT JOIN或RIGHT JOIN查询）就行了。比如说，当需

要把数据表t1里的、在数据表t2里没有匹配的记录查找出来时，会写出一个如下所示的SELECT语句：

```
SELECT t1.* FROM t1 LEFT JOIN t2 ON t1.id = t2.id WHERE t2.id IS NULL;
```

相应地，从数据表t1里找出并删除这些记录的DELETE语句也要使用一个LEFT JOIN：

```
DELETE t1 FROM t1 LEFT JOIN t2 ON t1.id = t2.id WHERE t2.id IS NULL;
```

MySQL 4.0.2及以后的版本还支持一种稍微不同的涉及多个数据表的DELETE语法。这种语法使用一个FROM子句来列举删除动作所涉及的数据表，使用一个USING子句来列举删除条件所涉及的数据表。前面那几个涉及多个数据表的DELETE语句可以用这种语法改写为如下所示的样子：

```
DELETE FROM t1 USING t1, t2 WHERE t1.id = t2.id;
DELETE FROM t1, t2 USING t1, t2 WHERE t1.id = t2.id;
DELETE FROM t1 USING t1 LEFT JOIN t2 ON t1.id = t2.id WHERE t2.id IS NULL;
```

涉及多个数据表的删除操作还可以利用外键机制来实现，其基本思路是在各有关数据表之间建立一个带有ON DELETE CASCADE约束条件的外键关系。这一思路的具体做法参见第3.8节。

用来编写涉及多个数据表的DELETE语句的基本原则和做法也适用于涉及多个数据表的UPDATE语句的编写工作：先把本次操作将要涉及到的数据表都列举出来；再对数据列的名字进行必要的限定。我们来看一个例子：在于2002年9月23日对学生们进行的测验中，有一道选择题没有一位同学回答正确，后来发现这是因为这道题的标准答案错了。结果，必须给每位同学都加1分。如果无法使用涉及多个数据表的UPDATE语句，就得用两条语句才能完成这项工作。首先，根据给定日期把该次测验的事件ID查出来：

```
SELECT @id := event_id FROM event WHERE date = '2002-09-23' AND type = 'Q';
```

然后，再通过这个事件ID去检索和修改学生们的测验成绩：

```
UPDATE score SET score = score + 1 WHERE event_id = @id;
```

如果允许使用涉及多个数据表的UPDATE语句，只要用一条语句即可完成这项工作：

```
UPDATE score, event SET score.score = score.score + 1
WHERE score.event_id = event.event_id
AND event.date = '2002-09-23' AND event.type = 'Q';
```

利用涉及多个数据表的UPDATE语句，不仅能根据另一个数据表的内容去修改某数据表里的记录，还能从一个数据表里把一些数据列值拷贝到另一个数据表里去。比如说，下面这条语句将从数据表t1里把那些其id值在数据表t2里有匹配的记录的t1.a数据列值拷贝到数据表t2里的t2.a数据列里去。

```
UPDATE t1, t2 SET t2.a = t1.a WHERE t2.id = t1.id;
```

### 3.7 事务处理

所谓的“事务”(transaction)其实就是一组SQL语句，但这组SQL语句在执行时将被视为

一个不可中断的逻辑单元。事务处理机制的作用之一是确保有关数据记录在本次事务执行期间不会被其他客户（程序）修改。MySQL能够自动锁定单条SQL语句，使客户程序不会互相干扰。（比如说，两个客户程序不能在同一时刻去修改数据表里的同一条记录。）但这种单条SQL语句的自动锁定机制并不足以保证数据库操作总能达到预期的效果，因为有些数据库操作必须经由多条SQL语句才能完成。在这类场合，不同的数据库操作仍有可能互相干扰。在一个多客户（程序）环境里，事务处理机制能把多条SQL语句归组为一个不可中断的执行单元，从而防止了在多个客户程序环境中有可能发生的并发问题。

事务处理还包括提交和回滚功能，这些功能允许你要求语句必须是作为一个单元来执行或者根本不是作为一个单元来执行。也就是说，如果事务处理成功了，构成本次事务的各条语句将全部执行成功；如果本次事务的某个部分失败了，那么此前各语句的执行效果都将被撤销，数据库将被回滚为本次事务开始之前的状态。

事务处理系统的典型特点是具备ACID特征。ACID是Atomic（原子的）、Consistent（一致的）、Isolated（隔离的）以及Durable（持续的）四个单词的首字母缩写，它们代表着事务处理应该具备的四个特征：

- 原子性。组成事务处理的语句形成了一个逻辑单元，不能只执行其中的一部分。
- 一致性。在事务处理执行之前和之后，数据库是一致的。换句话说，事务处理不会搞乱数据库。
- 隔离性。一个事务处理对另一个事务处理没有影响。
- 持续性。当事务处理成功执行到结束的时候，其效果在数据库中被永久记录下来。

有些MySQL数据表类型（如ISAM、MyISAM、HEAP）是不支持事务处理机制的，有些（如BDB和InnoDB）则支持事务处理机制。有些操作必须通过事务处理机制来实现，如果不注意这些问题，就会遇到一些麻烦——本节将对这类麻烦进行演示并对实现这类操作的事务化手段和非事务化手段进行介绍。

### 3.7.1 事务处理机制的用途

当有多个客户程序试图对同一个数据库进行修改而每个改变又需要使用多条语句才能完成的时候，就可能导致各种并发问题。我们来看一个例子：假设你正从事着服装销售行业；售货员每完成一笔交易，你的销售记账软件就会自动刷新库存记录。于是，当同时进行多次销售时，就会发生以下操作事件。在这个例子里，假设最初的衬衫库存数是47件。

1) 售货员A卖出了3件衬衫并登记了这次销售。为了刷新数据库里的库存数据，销售记账软件先要把当前库存衬衫数（47件）选取出来，如下所示：

```
SELECT quantity FROM inventory WHERE item = 'shirt';
```

2) 与此同时，售货员B也卖出了2件衬衫并登记了这次销售。为了刷新数据库里的库存数据，第二个收银台里的销售记账软件也先要把当前库存衬衫数（47件）选取出来，如下所示：

```
SELECT quantity FROM inventory WHERE item = 'shirt';
```

3) 第一台收银机计算出新库存数将是 $47-3=44$ ，然后对衬衫库存数做了相应的修改：

```
UPDATE inventory SET quantity = 44 WHERE item = 'shirt';
```

4) 第二台收银机计算出新库存数将是 $47-2=45$ ，然后对衬衫库存数也做了相应的修改：

```
UPDATE inventory SET quantity = 45 WHERE item = 'shirt';
```

作为两次销售事件的结果，你的售货员应该卖出5件衬衫（这是正确的），可数据库里的库存记录却告诉你说还有45件库存（这就有问题了，因为它应该等于42件）。之所以会出现这样的问题，是因为整个操作必须使用两条语句才能完成：第一条用来查知当前的库存，第二条用来刷新库存数据；第二条语句必须依赖于第一条语句所检索到的库存值。当这两组多语句操作在时间上发生重叠时，分属两组操作的不同语句就会彼此交织并相互干扰。要想解决这个问题，就必须保证两组（甚至更多组）操作的语句在执行时不会彼此干扰。事务处理机制就是为了实现这一目的而被提出来的，它将把每位售货员用来完成记账操作的多条语句当做一个不可分割的单元来执行，确保它们不会彼此干扰。这样一来，在售货员A的记账操作没有结束之前，售货员B的记账操作语句就不会开始执行。

与需要多条语句才能完成一次数据库处理操作有关的另一个问题是：如果处理不当，在执行过程中意外出错的某次操作可能会把你的数据库留在一个修改了一半（即前后不一致）的状态。把钱款从一个账户转入另一个账户的资金划拨过程就是一个典型的例子。例如，假设比尔给鲍伯开出了一张100美元的转账支票而鲍伯打算兑现这张支票。此时，比尔的账户应该减少100美元，而鲍伯的账户应该增加100美元：

```
UPDATE account SET balance = balance - 100 WHERE name = 'Bill';
```

```
UPDATE account SET balance = balance + 100 WHERE name = 'Bob';
```

如果系统在上面这两条语句的执行过程中崩溃了，这次转账操作就将半途而废。如果系统不具备事务处理机制的话，你就只能依靠手工分析操作日志的办法才能搞清楚这次转账操作在故障发生时进行到了哪一步并决定是整个地撤销这次操作还是完成它。事务处理机制中的回滚功能将使你对这类问题做出正确的处理——它将撤销故障发生之前已经执行了的各有关语句的效果，把数据库恢复为事务开始之前的样子。（虽然仍需要由你去判断哪些事务尚未被记录到数据库里并再次提交它们，但你至少用不着再担心那些半途而废的事务会把你的数据库弄得一团糟了。）

### 3.7.2 事务问题的非事务实现办法

在不具备事务处理机制的环境里，与事务有关的问题有些能得到解决，有些则不能。下面的讨论将告诉你：在不使用事务处理机制的前提下，与事务有关的问题哪些是能够得到解决的，哪些是无法解决的。为了对付这类场合中的并发问题，可以采取以下几种方案：

- **明确地锁定各有关数据表。**这个方案的思路是：把各有关语句归纳为一个执行单元，并在它们的前、后相应地加上一些LOCK TABLES和UNLOCK TABLES语句。先把将会用到的各有关数据表全部锁定，然后提交查询，最后再解除（释放）这些锁定。这样，在你锁定各有关数据表期间，其他人将无法对它们进行修改。下面是我们采用数据表锁定方案而实现的库存修改操作：



- 1) 售货员A卖出了3件衬衫并登记了这次销售。在刷新库存数据的时候,销售记账软件将先申请一个数据表锁,然后再把当前库存衬衫数(47件)检索出来,如下所示:

```
LOCK TABLES inventory WRITE;
SELECT quantity FROM inventory WHERE item = 'shirt';
```

因为最终目标是修改库存数据表,即对之进行写操作,所以这里必须进行WRITE锁定。

- 2) 与此同时,售货员B也卖出了2件衬衫并登记了这次销售。在刷新库存数据的时候,第二个收银台里的销售记账软件也得先申请一个数据表锁,如下所示:

```
LOCK TABLES inventory WRITE;
```

此时,因为售货员A已经锁定了这个数据表,所以这条语句将被阻塞。

- 3) 第一台收银机计算出新库存是 $47-3=44$ ,并由此对衬衫库存数做了相应的修改,然后解除了对数据表的锁定,如下所示:

```
UPDATE inventory SET quantity = 44 WHERE item = 'shirt';
UNLOCK TABLES;
```

- 4) 当第一台收银机解除了对数据表的锁定之后,第二台收银机将成功地锁定那个数据表,并正确地检索出当前的衬衫库存数(44件),如下所示:

```
SELECT quantity FROM inventory WHERE item = 'shirt';
```

- 5) 第二台收银机计算出新库存是 $44-2=42$ ,并由此对衬衫库存数做了相应的修改,然后解除了对数据表的锁定,如下所示:

```
UPDATE inventory SET quantity = 42 WHERE item = 'shirt';
UNLOCK TABLES;
```

现在,分属两次记账操作的各有关语句没有混杂在一起,从而得到了正确的衬衫库存数。如果需要用到多个数据表,就必须在执行有关查询之前把它们全都锁定。不过,如果只需从某个特定的数据表读取数据,就只需对该数据表进行读锁定就可以了,没有必要对它进行写锁定。(读锁定允许别的客户程序在你使用该数据表的期间从该数据表读取数据,但不允许他们对该数据表进行写操作。)假定你有一些用来对inventory数据表进行修改的查询命令,同时还需要从customer数据表读取一些数据。在这种情况下,就可以对inventory数据表进行写锁定,对customer数据表进行读锁定,如下所示:

```
LOCK TABLES inventory WRITE, customer READ;
... use the tables here ...
UNLOCK TABLES;
```

- 使用相对修改语句而不是绝对修改语句。在采用明确锁定各有关数据表的办法而实现的库存刷新操作里,我们得先用一条语句来查出当前的库存记录,根据卖出了几件衬衫来计算出新的库存数,然后再用另一条语句把库存数刷新为新值。避免多个客户(程序)的操作彼此干扰的另一种办法是把有关操作压缩为一条语句,即从消除多语句操作中前、后语句的依赖现象入手。注意,并非每一种多语句操作都能被压缩为一条语句,但就库存刷新例子而言,这个方案还是切实可行的。只要简单地使用一条参照当前库存值的UPDATE语句,



就可以把衬衫库存数据的刷新操作压缩为一条步骤：

1) 售货员A卖出了3件衬衫，销售记账软件把衬衫的库存数减去3：

```
UPDATE inventory SET quantity = quantity - 3 WHERE item = 'shirt';
```

2) 售货员B卖出了2件衬衫，销售记账软件再把衬衫的库存数减去2：

```
UPDATE inventory SET quantity = quantity - 2 WHERE item = 'shirt';
```

如果采用的是这一方案，因为数据库的修改操作不再需要用多条语句来完成，所以并发问题也就消除了。这意味着根本不必去明确地锁定各有关数据表。换句话说，如果你打算完成的操作与这个例子的情况相类似，那根本不需要求助于事务处理机制。

非事务化解决方案可以成功地应用在很多类型的问题上，但它们也有不足之处，主要表现在：

- 并非每一种操作都能被改写为相对修改语句。有些问题只能用多条语句来解决——此时，就必须考虑并发问题并解决好它们。
- 在一个多语句操作的执行期间锁定有关的数据表可以避免客户（程序）之间的相互干扰，可你知道万一这个操作半路出错了会发生什么样的事情吗？在发生这类问题的时候，你肯定希望能够把已经执行了的语句的效果全部撤销掉以避免把数据库留在一种只有部分数据得到修改的不一致状态。

然而，虽然明确地锁定各有关数据表可以解决并发问题，但它对半路出错的操作却提供不出任何补救措施。

- 明确地锁定各有关数据表的方案要求你必须亲自对各有关数据表进行锁定和解除锁定。如果该操作所涉及的数据表发生了变化，也必须由你亲自对那些LOCK TABLES语句做出相应的修改。与此形成鲜明对照的是，支持事务处理机制的数据库系统能够自行决定需要锁定哪些数据表并完成相应的锁定。

事务处理机制能够帮助你解决所有这些问题。首先，事务处理程序把一组语句整个地当作一个单元来执行，从根本上避免了客户（程序）之间的相互干扰，并发问题也就迎刃而解。其次，即使出现了意外，事务处理机制也能对那些半途而废的事务进行回滚，使它们不会对你的数据库造成损害。再有，事务处理机制能自动完成对各有关数据表的必要的锁定。

### 3.7.3 利用事务处理机制来保证语句的安全执行

使用事务处理机制的前提是必须选用一种支持事务处理机制的数据表类型。ISAM、MyISAM和HEAP数据表类型都不支持事务处理；只能选用BDB或InnoDB数据表。BDB和InnoDB数据表处理程序分别始见于MySQL 3.23.17和3.23.29版本的二进制发行版本，并且从MySQL 3.23.34版本开始被添加到了源代码发行版本里。不过，要想最大限度地用好事务处理机制，建议大家尽可能地选用最新的发行版本。如果你拿不准自己的MySQL服务器是否包含有BDB或InnoDB数据表处理程序，可以参阅第3.4.1节。

在默认情况下，MySQL运行在自动提交模式下，即每条语句做出的修改将被立即提交到数据库里生效。从效果上讲，每条语句都可以看做是一个事务。如果想明确地进行事务处理，就

需要禁用自动提交模式并告诉MySQL何时去提交或回滚那些修改。

进行事务处理的做法之一是这样的：先用BEGIN语句来禁用自动提交模式，然后依次执行构成这次事务的各条语句，最后用COMMIT语句来结束事务并使修改生效。如果在事务处理过程中发生了错误，则发出一条ROLLBACK语句来撤销那些修改。BEGIN语句只是在本次事务期间暂时禁用自动提交模式，在提交或者回滚了本次事务之后，自动提交模式将恢复为发出BEGIN语句之前的设置状态——如果自动提交模式原本处于激活状态，本次事务结束后将重新回到自动提交模式；如果它原本处于禁用状态，本次事务结束后可以直接开始下一次事务。

下面是一个使用了这种做法的事务处理示例。我们先来创建一个数据表：

```
mysql> CREATE TABLE t (name CHAR(20), UNIQUE (name)) TYPE = INNODB;
```

上面这条语句创建了一个InnoDB数据表，也可以创建一个BDB数据表。接下来，我们用BEGIN语句开始一次事务，给这个数据表增加几个数据行，再提交这个事务，然后看看数据表变成了什么样子：

```
mysql> BEGIN;
mysql> INSERT INTO t SET name = 'William';
mysql> INSERT INTO t SET name = 'Wallace';
mysql> COMMIT;
mysql> SELECT * FROM t;
+-----+
| name  |
+-----+
| Wallace |
| William |
+-----+
```

可以看到，数据行都已经记录在了数据表里。如果你事先启动了另一个mysql客户程序的实例，并在插入这些数据行之后、提交这个事务之前去选取数据表t的内容，是不会看到这些数据行的——只有在第一个mysql进程执行完COMMIT语句之后，才能在第二个mysql进程里看到数据表t里的数据行。

如果在某次事务处理期间发生了错误，可以用ROLLBACK语句来撤销它。仍以数据表t为例，下面这些语句将让你看到事务处理机制中的回滚功能：

```
mysql> BEGIN;
mysql> INSERT INTO t SET name = 'Gromit';
mysql> INSERT INTO t SET name = 'Wallace';
ERROR 1062: Duplicate entry 'Wallace' for key 1
mysql> ROLLBACK;
mysql> SELECT * FROM t;
+-----+
| name  |
+-----+
| Wallace |
| William |
+-----+
```

第二条INSERT语句试图把一个数据行插入到数据表t里去，但它的name值与数据表t里某个现有的数据行发生了重复。因为name数据列上有一个UNIQUE索引，所以这条INSERT语句将执行失败。在发出ROLLBACK语句之后，数据表t将只包含有本次事务开始之前的那两个数据行——也就是说，在错误发生之前进行的INSERT操作被撤销了，它的修改效果没有记录到数据表里去。

在某次事务处理期间发出的另一条BEGIN语句将（隐含地）立刻提交当前事务并开始一次新的事务。

进行事务处理的另一种做法是直接使用SET语句来改变自动提交模式：

```
SET AUTOCOMMIT = 0;
SET AUTOCOMMIT = 1;
```

把AUTOCOMMIT设置为0将禁用自动提交模式，随后的语句都将成为当前事务的组成部分——直到你发出一条COMMIT或ROLLBACK语句来提交或撤销这次事务为止。SET语句对自动提交模式的改变是“永久性”的——自动提交模式在你把AUTOCOMMIT重新设置为1之前将一直保持在禁用状态，所以结束前一次事务的同时也立刻开始了下一次事务。还可以用重新激活自动提交模式的办法来提交当前事务。

下面是一个使用了这种方法的事务处理示例，我们仍将使用上一个例子中的数据表：

```
mysql> DROP TABLE t;
mysql> CREATE TABLE t (name CHAR(20), UNIQUE (name)) TYPE = INNODB;
```

接下来，禁用自动提交模式，插入一些记录，然后提交这次事务：

```
mysql> SET AUTOCOMMIT = 0;
mysql> INSERT INTO t SET name = 'William';
mysql> INSERT INTO t SET name = 'Wallace';
mysql> COMMIT;
mysql> SELECT * FROM t;
```

name
Wallace
William

现在，两条记录已经被提交到数据表里了，但自动提交模式却仍保持在禁用状态。如果你继续发出一些语句，它们将成为一个新事务的组成部分，而这个新事务的提交或回滚与第一个事务没有任何依赖关系。下面这些语句能让我们看清：1）自动提交模式仍保持在禁用状态；2）ROLLBACK将撤销尚未被提交的语句：

```
mysql> INSERT INTO t SET name = 'Gromit';
mysql> INSERT INTO t SET name = 'Wallace';
ERROR 1062: Duplicate entry 'Wallace' for key 1
mysql> ROLLBACK;
mysql> SELECT * FROM t;
```

```

+-----+
| name   |
+-----+
| Wallace|
| William|
+-----+

```

如果想激活自动提交模式，就必须使用下面这条语句：

```
SET AUTOCOMMIT = 1;
```

以下情况也将结束当前事务：

- 除SET AUTOCOMMIT、BEGIN、COMMIT和ROLLBACK等明显会对事务处理产生影响的语句外，有些语句也会隐含地结束一次事务——因为它们不能成为事务的一个组成部分。如果在事务期间发出了下列语句之一，MySQL服务器就会在执行该语句之前先提交当前事务。能够引起提交动作的语句如下所示：

```

ALTER TABLE
CREATE INDEX
DROP DATABASE
DROP INDEX
DROP TABLE
LOAD MASTER DATA
LOCK TABLES
RENAME TABLE
TRUNCATE TABLE
UNLOCK TABLES (如果数据表已经处于锁定状态)

```

- 如果客户连接在某次事务被提交之前结束了或者意外断开，MySQL服务器将自动回滚这次事务。

事务处理机制在各种场合都有用武之地。举个例子，假设你在考试记分项目的score数据表里发现有两位学生的成绩颠倒了，现在需要把它们调整过来。录错的两个分数如下所示：

```
mysql> SELECT * FROM score WHERE event_id = 5 AND student_id IN (8,9);
```

student_id	event_id	score
8	5	18
9	5	13

这里，需要把编号为8的学生成绩改为13，把编号为9的学生成绩改为18。这些修改可以用下面两条语句轻松实现：

```

UPDATE score SET score = 13 WHERE event_id = 5 AND student_id = 8;
UPDATE score SET score = 18 WHERE event_id = 5 AND student_id = 9;

```

注意，这两条语句必须同时执行成功才有意义——而这正是事务处理机制大显身手的地方。我们先通过BEGIN语句来解决这一问题：

```
mysql> BEGIN;
mysql> UPDATE score SET score = 13 WHERE event_id = 5 AND student_id = 8;
mysql> UPDATE score SET score = 18 WHERE event_id = 5 AND student_id = 9;
mysql> COMMIT;
```

我们再通过SET语句（即明确地设置自动提交模式的方法）来完成同样的工作：

```
mysql> SET AUTOCOMMIT = 0;
mysql> UPDATE score SET score = 13 WHERE event_id = 5 AND student_id = 8;
mysql> UPDATE score SET score = 18 WHERE event_id = 5 AND student_id = 9;
mysql> COMMIT;
mysql> SET AUTOCOMMIT = 1;
```

不管采用的是哪一种方法，其结果应该是一样的——两位学生的成绩正确地交换了过来：

```
mysql> SELECT * FROM score WHERE event_id = 5 AND student_id IN (8,9);
```

student_id	event_id	score
8	5	13
9	5	18

### 3.8 外键与引用完整性

外键关系允许把某一个数据表里的一个索引声明为与另一个数据表里的一个索引有关联关系，还允许对外键所在的数据表设置一些操作处理方面的约束条件。数据库将根据外键关系所定义的规则来维护引用完整性（referential integrity）。比如说，样板数据库sampdb中的score数据表里有一个student\_id数据列，用来将学生们的考试成绩与数据表student中的各位学生关联起来。在第1章里创建这些数据表的时候没有在它们之间建立任何关联关系。现在看来，应该把数据列score.student\_id声明为数据列student.student\_id的一个外键。这样，当我们往score数据表里插入一个考试分数记录的时候，如果其中的student\_id值在student数据表里不存在，这条记录就不会被插入到score数据表里去。（换句话说，这个外键将确保我们不会为一个并不存在的学生录入考试分数。）我们还可以设置一个这样的限定条件：当我们从student数据表里删除一位学生的时候，score数据表里与这位学生有关的考试分数记录将同时被自动删除。通常把这种删除操作称为级联删除（cascaded delete），因为这种删除操作的实际效果是从一个数据表逐步推进到另一个数据表里。

外键有助于保持数据的一致性，用起来相当方便。如果没有外键，你就得另想办法去记住数据表之间的依赖关系并在你的应用程序里保持有关数据的一致性。在很多场合，这一目标并不难实现，通常只需在应用程序里增加一些DELETE语句就能达到目的——当你从某个数据表删除数据记录的时候，这些“额外的”DELETE语句将确保与该数据表相关联的所有其他数据表里的有关记录也都会被删除掉。不过，如果数据表之间有着比较复杂的关系，让应用程序采用这种额外增加一些DELETE语句的办法来维持它们之间的依赖关系就难免会力不从心了。此外，既然数据库引擎能够替你进行一致性检查，为什么不让它去做呢？



MySQL里的外键支持是由InnoDB数据表处理程序提供的。在这一节里，将介绍如何建立InnoDB数据表、如何定义外键、以及外键对数据表的使用会有哪些影响。我们先来熟悉几个术语：

- 父表 (parent table) 是包含着原始键值的数据表。
- 子表 (child table) 是引用了父表中的键值的数据表，这种引用使它与父表构成了依赖关系。
- 父表和子表是通过父表中的键值而关联在一起的。具体说来，就是子表中的索引将引用父表中的索引，它的值或者必须与父表中的某个值相匹配，或者将被设置为NULL以表明父表中没有与之对应的记录。子表中的索引就是我们所说的外键——这个索引存在于父表的外部，但包含的值却指向父表。此外，外键关系可以设置成不允许出现NULL值的情况，此时，每一个外键值都必须与父表中的某个值相匹配。

InnoDB数据表将通过这些规则来保证外键关系不受到“没有匹配”或“匹配错误”的影响，而这称为“引用完整性”。

下面是在子表中声明一个外键的语法，方括号中的内容是可选的：

```
FOREIGN KEY [index_name] (index_columns)
    REFERENCES tbl_name (index_columns)
    [ON DELETE action]
    [ON UPDATE action]
    [MATCH FULL | MATCH PARTIAL]
```

注意，虽然MySQL会对这个语法的所有组成部分进行词法分析，但InnoDB并没有把所有子句的语义都实现出来。MySQL目前还不支持这个语法中的ON UPDATE 和MATCH子句——即使你写出了这两个子句，它们也会被忽略<sup>①</sup>。

下面是在定义InnoDB数据表时要特别注意的几个地方：

- FOREIGN KEY后面的index\_columns部分列出了构成子表中的这个索引的各有关数据列，它们必须与父表中的索引值相匹配。index\_name则无关紧要，即使给出，也会被忽略。
- REFERENCES子句中的tbl\_name和index\_columns分别给出了父表的名字和与子表中的外键相对应的父表索引数据列的名字。REFERENCES子句里的index\_columns部分必须与关键字FOREIGN KEY后面的index\_columns部分有相同数目的数据列。
- ON DELETE用来设定在父表记录被删除时子表将发生什么情况。允许使用的操作动作如下所示：
  - ON DELETE CASCADE：当父表中的某个记录被删除时，子表中与之匹配的记录也将被删除。从效果上讲，这等于是对父表和子表进行级联删除操作。也就是说，如果删除操作涉及多个数据表，只需在父表中删除有关的数据行就行了，子表中的数据行删除操作将由InnoDB负责完成。
  - ON DELETE SET NULL：当父表中的某个记录被删除时，子表中与之匹配的记录中的索引数据列将被设置为NULL。如果使用了这个选项，就必须把在外键定义中列出的子表数据列全都声明为允许取值为NULL值。（使用了这个选项的另一层含义是不能把外

① 对于其他非InnoDB的数据表类型，FOREIGN KEY定义将整个地被词法分析器忽略。

键声明为PRIMARY KEY, 因为主键值不允许取值为NULL值。)

在定义外键关系的时候, 一定要遵守以下规则:

- 子表中必须有这样一个索引: 外键数据列将出现在这个索引的最前面——也就是说, 在定义这个索引的时候, 必须首先列出外键数据列。父表中则必须有这样一个索引: 出现在外键定义里的REFEREBCES子句中的数据列将出现在这个索引的最前面——也就是说, 在定义这个索引的时候, 必须首先列出那些将出现在有关的REFEREBCES子句中的数据列。(换句话说, 构成外键关系的父表数据列和子表数据列在它们各自所在的数据表里都必须被索引。)在父表和子表里对有关数据列进行索引的工作必须由你明确地完成, InnoDB是不会创建它们的。
- 父表索引和子表索引中相互对应的数据列必须是兼容的类型。比如说, 不能用一个INT数据列去匹配一个CHAR数据列。相互对应的字符型数据列必须是同样的长度。相互对应的整数型数据列不仅大小要相同(即同为INT、MEDIUMINT或BIGINT等等), 还必须同时是或不是UNSIGNED。

下面结合一个例子来看看这一切都是如何工作的。从创建两个名字分别是parent和child的数据表开始, child数据表里包含着一个外键, 这个外键引用了parent数据表里的par\_id数据列:

```
CREATE TABLE parent
(
    par_id      INT NOT NULL,
    PRIMARY KEY (par_id)
) TYPE = INNODB;

CREATE TABLE child
(
    par_id      INT NOT NULL,
    child_id    INT NOT NULL,
    PRIMARY KEY (par_id, child_id),
    FOREIGN KEY (par_id) REFERENCES parent (par_id) ON DELETE CASCADE
) TYPE = INNODB;
```

在这个例子里, 我们在定义外键的时候使用了ON DELETE CASCADE。于是, 当你从parent数据表里删除一个记录的时候, 与该记录的par\_id值相匹配的child数据表里的有关记录将自动地被删除掉。

下面, 我们先往parent数据表里插入几条记录, 再相应地往child数据表里添加一些带有匹配键值的记录:

```
mysql> INSERT INTO parent (par_id) VALUES(1),(2),(3);
mysql> INSERT INTO child (par_id,child_id) VALUES(1,1),(1,2);
mysql> INSERT INTO child (par_id,child_id) VALUES(2,1),(2,2),(2,3);
mysql> INSERT INTO child (par_id,child_id) VALUES(3,1);
```

这些语句将产生如下所示的数据表内容。注意, child数据表里的每一个par\_id值都匹配着parent数据表里的一个par\_id值:

```
mysql> SELECT * FROM parent;
```

```
+-----+
| par_id |
+-----+
|      1 |
|      2 |
|      3 |
+-----+
```

```
mysql> SELECT * FROM child;
```

```
+-----+-----+
| par_id | child_id |
+-----+-----+
|      1 |      1 |
|      1 |      2 |
|      2 |      1 |
|      2 |      2 |
|      2 |      3 |
|      3 |      1 |
+-----+-----+
```

怎样才能证明InnoDB确实在插入操作中对外键关系进行了检查呢？试着用INSERT语句往child数据表里插入一条记录，但这条记录的par\_id值在parent数据表里并不存在：

```
mysql> INSERT INTO child (par_id,child_id) VALUES(4,1);
ERROR 1216: Cannot add a child row: a foreign key constraint fails
```

现在，再来看看在父表里删除一条记录时会发生什么情况：

```
mysql> DELETE FROM parent where par_id = 1;
```

这条DELETE语句从parent数据表里删除了给定的记录，如下所示：

```
mysql> SELECT * FROM parent;
```

```
+-----+
| par_id |
+-----+
|      2 |
|      3 |
+-----+
```

同时，这条DELETE语句还对child数据表里的有关记录进行了级联删除，如下所示：

```
mysql> SELECT * FROM child;
```

```
+-----+-----+
| par_id | child_id |
+-----+-----+
|      2 |      1 |
|      2 |      2 |
|      2 |      3 |
|      3 |      1 |
+-----+-----+
```

从上面这个例子可以清楚地看出：当在父表里删除一条记录的时候，子表中与之对应的有关记录也都被删掉了。另一种做法是让那些子表记录仍保留在数据表里，但把它们的外键数据列设置为NULL——这需要对child数据表的定义做三处修改：

- 用ON DELETE SET NULL代替ON DELETE CASCADE。这将使InnoDB把有关的外键数据列（par\_id）设置为NULL而不是把那些记录删除掉。
- child数据表的原始定义把par\_id数据列声明为NOT NULL——这与ON DELETE SET NULL的使用要求当然不相符合，所以我们还得把这个数据列声明为NULL。
- child数据表的原始定义还把par\_id数据列声明为某个PRIMARY KEY的组成部分。但是，PRIMARY KEY不允许包含NULL值，所以，既然允许par\_id数据列里出现NULL值，我们就得把PRIMARY KEY改为UNIQUE索引。InnoDB数据表中的UNIQUE索引必须具备惟一性，但NULL值是个例外，它可以在索引里出现多次。

要想检验上述修改的效果，得先用parent数据表的原始定义重新创建该数据表并把原来的那些记录再次加载到其中，然后再用如下所示的新定义来创建一个新的child数据表：

```
CREATE TABLE child
(
    par_id      INT NULL,
    child_id    INT NOT NULL,
    UNIQUE (par_id, child_id),
    FOREIGN KEY (par_id) REFERENCES parent (par_id) ON DELETE SET NULL
) TYPE = INNODB;
```

在插入新记录的时候，新child数据表有着与刚才同样的表现：只允许插入其par\_id值在父表里已经存在的记录，不允许插入其par\_id值在父表里不存在的记录<sup>①</sup>。

```
mysql> INSERT INTO child (par_id,child_id) VALUES(1,1),(1,2);
mysql> INSERT INTO child (par_id,child_id) VALUES(2,1),(2,2),(2,3);
mysql> INSERT INTO child (par_id,child_id) VALUES(3,1);
mysql> INSERT INTO child (par_id,child_id) VALUES(4,1);
ERROR 1216: Cannot add a child row: a foreign key constraint fails
```

新、旧child数据表的行为差异主要体现在删除父表记录的时候。我们先删除一个父表记录，然后检查一下child数据表的内容，看看会发生什么情况：

```
mysql> DELETE FROM parent where par_id = 1;
mysql> SELECT * FROM child;
+-----+-----+
| par_id | child_id |
+-----+-----+
| NULL   | 1       |
| NULL   | 2       |
| 2      | 1       |
```

① 严格地讲，新、旧child数据表在插入记录时还是有一点细微差别的。因为child数据表的新定义把par\_id数据列声明为NULL，所以现在可以把一个包含着NULL值的记录插入其中而不会引起MySQL报告出错。

	2		2	
	2		3	
	3		1	
+-----+-----+				

可以看到，在child数据表里，par\_id数据列当初取值为1的两条记录并没有被删除掉，它们的par\_id数据列只是被设置为NULL值了，这正是ON DELETE SET NULL约束条件所规定的情况。

外键机制的各项功能并不是同时出现在MySQL里的，如下所示。最初的外键支持只能防止人们插入或者删除不符合外键约束条件的子表记录，其他功能是后来才增加的：

功 能	版 本 号
基本的外键支持	3.23.44
ON DELETE CASCADE	3.23.50
ON DELETE SET NULL	3.23.50

这个表隐含着这样一个提示：要想获得最完备的外键支持功能，最好使用MySQL 3.23.50或更新的版本。建议大家尽量使用最新版本的另一个理由是以下漏洞直到MySQL 3.23.50版本才得到了修正：

- 在MySQL 3.23.50之前的版本里，使用ALTER TABLE或CREATE INDEX语句去修改某个外键关系中的父InnoDB数据表或子InnoDB数据表都是不安全的，因为它们会解除父表和子表之间的外键约束条件。
- 在MySQL 3.23.50之前的版本里，SHOW CREATE TABLE语句显示不出外键定义。mysqldump程序也是如此，这使得很难从备份文件中正确地恢复出带有外键的数据表来。

### 没有外键支持怎么办

如果你的MySQL软件不具备InnoDB支持（因而无法使用外键），该如何去保持各有关数据表之间的引用完整性呢？

一般说来，通过应用程序逻辑来实现外键关系中的约束条件并不是很难。有时候，只需简单地调整一下数据录入流程就能解决问题。以考试记分项目中的student和score数据表为例，这两个数据表是隐含地通过各自的student\_id值而关联起来的。在考试或测验结束后，当需要把一组新的考试成绩录入到数据库里去的时候，为一名并不存在的学生而插入考试分数记录的事情是不太可能发生的。正确的考试分数录入流程应该是这样的：先用student数据表生成一份学生名单，再依照这份名单把每位学生的考分和ID编号生成一条新的score数据表记录。采用这一流程，给一位并不存在的学生录入了一个成绩的事情可以说是不可能发生的——因为你不太可能会臆造出一个考试分数记录并把它插入到score数据表里去。

当从student数据表里删除一条学生记录时又会发生怎样的事情呢？假设有把编号为13的那位学生的记录删除掉。这意味着还得把这位学生的考试成绩记录全都删掉。如果有带级联删除动作的外键关系可供使用，只需简单地用下面这条语句把这位学生的student数据表记录删掉就



行了，MySQL将自动地把相应的score数据表记录全都删掉：

```
DELETE FROM student WHERE student_id = 13;
```

如果没有外键支持可用，就必须明确地发出如下所示的DELETE语句来模拟级联删除动作的效果，把各有关数据表里的有关记录全都删掉：

```
DELETE FROM student WHERE student_id = 13;
DELETE FROM score WHERE student_id = 13;
```

在MySQL 4及以后的版本里，还可以使用另一种方法来完成这一工作：用一条涉及多个数据表的DELETE语句来获得与级联删除动作同样的效果。但这里有一个容易忽视的陷阱要特别注意——下面这条DELETE语句似乎能够实现目标，只可惜它实际上并不正确：

```
DELETE student, score FROM student, score
WHERE student.student_id = 13 AND student.student_id = score.student_id;
```

这条语句的问题是这样的：如果这位学生还没有参加过任何考试或者测验（因而在score数据表里也就没有任何考试成绩记录），删除动作就将以失败告终；WHERE子句找不到符合条件的匹配，也就不会从student数据表里删除任何记录。这种场合正是LEFT JOIN（左关联）大显身手的地方——即便没有相匹配的score数据表记录，它也能从student数据表里找到给定的记录并把它删掉：

```
DELETE student, score FROM student LEFT JOIN score USING (student_id)
WHERE student.student_id = 13;
```

### 3.9 使用FULLTEXT全文本搜索

从3.23.23版本开始，MySQL又增加了一项对全文本进行搜索的功能。全文本搜索引擎使你能够在不使用模式匹配操作的前提下去搜索单词或短语。如果你想在给定的数据表上使用这一功能，就得先创建一个特殊类型的索引，这种索引具有以下特性：

- 全文本搜索是以FULLTEXT索引为基础的。这种索引只能在MyISAM数据表里创建，并且只能在TEXT数据列以及不带BINARY关键字的CHAR和VARCHAR数据列上创建。
- FULLTEXT搜索对字母的大小写不做要求，因为允许用来创建FULLTEXT索引的数据列类型（TEXT、CHAR、VARCHAR等）不要求字母的大小写。
- FULLTEXT搜索将忽略“常见”单词。这里“常见”意思是“至少在一半的数据表记录中都出现”。这一点很重要；当你打算建立一个测试性的数据表来体会一下FULLTEXT搜索功能时，千万要注意这个问题。（测试数据表至少要有三条记录。如果数据表里只有一条或者两条记录，那么，因为每个单词的出现频度都至少是50%，因此将得不到任何结果！）某些本身就十分常见的单词——比如“the”、“after”、“other”等——是FULLTEXT搜索的“盲点”，永远会被忽略。太短的单词也会被忽略。在默认情况下，这个“太短”被定义为少于四个字符；但如果你的MySQL服务器版本比较新，也许能把这个长度值设置得更小。
- 这里所说的“单词”指的是由字母、数字、撇号和下划线等字符构成的字符序列。根据这

一、定义，字符串“full-blooded”将被视为由“full”和“blooded”两个单词构成。一般说来，全文本搜索是对完整的单词而不是单词的片段进行匹配的，只要某条记录里包含有搜索字符串中的任何一个单词，FULLTEXT引擎就认为该记录与搜索字符串是匹配的。全文本搜索还有一种变体形式叫做“布尔全文本搜索”，它允许你给搜索动作再加上“所有单词都必须出现”的约束条件（对单词的先后出现顺序不做要求；但在搜索一个由多个单词组成的短语时，有关单词必须按它们在搜索字符串里的顺序依次出现才算匹配）。利用布尔搜索功能，还能对“不包含给定单词的记录”进行匹配，或者增加一个通配符来匹配那些以给定前缀开头的所有单词。

- FULLTEXT索引允许创建在一个或者多个数据列上。如果它创建在多个数据列上，基于这个索引的搜索动作就将在各有关数据列上同时进行。这种安排的负面效应是：不同的全文本搜索要求有专用的FULLTEXT索引，当进行一次全文本搜索的时候，在查询语句里写出的数据列清单必须与构成某个FULLTEXT索引的那些数据列完全吻合才行。比如说，如果有时要搜索数据列col1，有时要搜索数据列col2，有时又要同时搜索数据列col1和col2，这时就应该创建三个FULLTEXT索引：一个对应于col1，一个对应于col2，还有一个对应于col1加col2。

在上面介绍的这些特性里，有些功能需要在比较新的MySQL版本支持下才能正常工作。下表列出了哪种FULLTEXT功能最早出现在哪个MySQL版本：

功 能	版 本 号
基本的FULLTEXT搜索	3.23.23
可配置的参数	4.0.0
布尔搜索	4.0.1
短语搜索	4.0.2

下面是一些全文本搜索功能的用法示例。我们先创建出必要的FULLTEXT索引，然后再通过MATCH操作符用这些索引来完成查询。

FULLTEXT索引的创建方法与其他索引是一样的——既可以在当初创建数据表的时候用CREATE TABLE语句来定义之，也可以在数据表创建出来之后再通过ALTER TABLE或CREATE INDEX语句来添加之。我们知道，FULLTEXT索引要求使用MyISAM数据表，所以，如果在事先就已经知道自己创建的数据表将要用于FULLTEXT搜索，就可以利用MyISAM处理程序的这样一个特性：先创建数据表并填充好数据再添加索引的做法要比先创建好索引再填充数据的做法更快。假设有一个名为apothegm.txt的数据文件，这个文件的内容是一些如下所示的名人名言：

Aeschylus	Time as he grows old teaches many lessons
Alexander Graham Bell	Mr. Watson, come here. I want you!
Benjamin Franklin	It is hard for an empty bag to stand upright
Benjamin Franklin	Little strokes fell great oaks
Benjamin Franklin	Remember that time is money
Miguel de Cervantes	Bell, book, and candle

```

Proverbs 15:1      A soft answer turneth away wrath
Theodore Roosevelt Speak softly and carry a big stick
William Shakespeare But, soft! what light through yonder window breaks?

```

如果你打算分别对名人、名言或者是名人加名言进行搜索，就需要创建三个分别对应于名人、名言、以及名人加名言的FULLTEXT索引。以下语句将完成一个名为apothegm数据表的创建、填充数据和索引工作：

```

CREATE TABLE apothegm (attribution VARCHAR(40), phrase TEXT);
LOAD DATA LOCAL INFILE 'apothegm.txt' INTO TABLE apothegm;
ALTER TABLE apothegm
  ADD FULLTEXT (phrase),
  ADD FULLTEXT (attribution),
  ADD FULLTEXT (phrase, attribution);

```

建立起这个数据表之后，我们就可以对它进行全文本搜索了——MATCH负责列举将对之进行搜索的一个或者多个数据列，AGAINST()负责给出搜索字符串。如下所示：

```

mysql> SELECT * FROM apothegm WHERE MATCH(attribution) AGAINST('roosevelt');
+-----+-----+
| attribution          | phrase                                     |
+-----+-----+
| Theodore Roosevelt | Speak softly and carry a big stick |
+-----+-----+
mysql> SELECT * FROM apothegm WHERE MATCH(phrase) AGAINST('time');
+-----+-----+
| attribution          | phrase                                     |
+-----+-----+
| Benjamin Franklin  | Remember that time is money           |
| Aeschylus          | Time as he grows old teaches many lessons |
+-----+-----+
mysql> SELECT * FROM apothegm WHERE MATCH(attribution,phrase)
-> AGAINST('bell');
+-----+-----+
| attribution          | phrase                                     |
+-----+-----+
| Alexander Graham Bell | Mr. Watson, come here. I want you! |
| Miguel de Cervantes  | Bell, book, and candle               |
+-----+-----+

```

请注意最后一个查询示例，它把搜索字符串出现在不同数据列里的记录都查出来了；这正体现了FULLTEXT搜索能够同时对多个数据列进行搜索的特点。另一个值得注意的地方是有关的数据列在查询命令里是按(attribution, phrase)的顺序写出的，与我们在创建索引时给出的顺序(phrase, attribution)不一样；这正体现了单词在搜索字符串里的先后顺序对FULLTEXT搜索没有影响的特点。对FULLTEXT搜索来说，确保有一个FULLTEXT索引包含着你在查询命令里给出的数据列才是最关键的。

如果想知道某次搜索匹配到了多少条记录，可以使用COUNT(\*)函数，如下所示：

```
mysql> SELECT COUNT(*) FROM apothegm WHERE MATCH(phrase) AGAINST('time');
```

```
+-----+
| COUNT(*) |
+-----+
|         2 |
+-----+
```

在默认情况下，如果在WHERE子句里使用了MATCH表达式，FULLTEXT搜索的输出行将按照相关性逐渐递减的顺序排列。

“相关性”是用一些非负的浮点值来表示的，0表示“完全无关”。如果你想看到那些用来表示相关性的浮点值，可以像下面这样在输出列的清单里加上一个MATCH表达式：

```
mysql> SELECT phrase, MATCH(phrase) AGAINST('time') AS relevance
-> FROM apothegm;
```

```
+-----+-----+
| phrase                                     | relevance |
+-----+-----+
| Time as he grows old teaches many lessons | 1.1976701021194 |
| Mr. Watson, come here. I want you!       | 0 |
| It is hard for an empty bag to stand upright | 0 |
| Little strokes fell great oaks           | 0 |
| Remember that time is money               | 1.2109839916229 |
| Bell, book, and candle                    | 0 |
| A soft answer turneth away wrath          | 0 |
| Speak softly and carry a big stick        | 0 |
| But, soft! what light through yonder window breaks? | 0 |
+-----+-----+
```

在默认情况下，FULLTEXT搜索将把包含有任何一个被搜索单词的记录全都找出来。也就是说，下面这个查询将把包含有单词“hard”或者“soft”的记录都找出来：

```
mysql> SELECT * FROM apothegm WHERE MATCH(phrase)
-> AGAINST('hard soft');
```

```
+-----+-----+
| attribution | phrase |
+-----+-----+
| Benjamin Franklin | It is hard for an empty bag to stand upright |
| Proverbs 15:1     | A soft answer turneth away wrath |
| William Shakespeare | But, soft! what light through yonder window breaks? |
+-----+-----+
```

MySQL 4.0.1及以后的版本增加了对布尔模式的FULLTEXT搜索功能的支持，这使我们能够对涉及多个单词的匹配动作做进一步的控制。要想进行这种搜索，就得在AGAINST()函数中的搜索字符串的后面加上“IN BOOLEAN MODE”关键字。布尔搜索具有下列特性：

- 与“常见”单词有关的50%规则将失去效力——即使某个单词在超过一半的记录里都有出现，它也会被搜索出来。
- 搜索结果将不再根据相关性进行排序。

- 允许给搜索字符串中的单词加上一些修饰符。前导的加号(+)或减号(-)表示该单词必须出现或者不得出现在匹配记录里。比如说,搜索字符串'bell'匹配的是那些包含有单词“bell”的记录,但布尔模式下的搜索字符串'+bell -candle'匹配的却是那些包含有单词“bell”但不包含单词“candle”的记录。如下所示:

```
mysql> SELECT * FROM apothegm
      -> WHERE MATCH(attribution,phrase)
      -> AGAINST('bell');
```

attribution	phrase
Alexander Graham Bell	Mr. Watson, come here. I want you!
Miguel de Cervantes	Bell, book, and candle

```
mysql> SELECT * FROM apothegm
      -> WHERE MATCH(attribution,phrase)
      -> AGAINST('+bell -candle' IN BOOLEAN MODE);
```

attribution	phrase
Alexander Graham Bell	Mr. Watson, come here. I want you!

被搜索单词尾缀的星号(\*)起着通配符的作用,任何包含着以被搜索单词开头的某个单词的记录都将与之匹配。比如说,'soft\*'将匹配“soft”、“softly”、“softness”等单词,如下所示:

```
mysql> SELECT * FROM apothegm WHERE MATCH(phrase)
      -> AGAINST('soft*' IN BOOLEAN MODE);
```

attribution	phrase
Proverbs 15:1	A soft answer turneth away wrath
William Shakespeare	But, soft! what light through yonder window breaks?
Theodore Roosevelt	Speak softly and carry a big stick

注意,这种通配符机制不能匹配长度小于索引单词最短长度的单词。

完整的修饰符清单可以在附录C的MATCH条目下查到。

- 类似于非布尔模式的全文本搜索情况,“盲点”单词仍将被忽略——哪怕用加号修饰符把它们标记为必须出现也是如此。比如说,搜索字符串'+Alexander +the +great'将把包含着单词“Alexander”和“great”的记录找出来,但盲点单词“the”却会被忽略掉。
- 在对由多个单词组成的某个短语进行搜索的时候,可以要求被搜索单词必须按指定的顺序出现。这种短语搜索要求MySQL必须是4.0.2或更高的版本。必须把搜索字符串用双引号引起来,还可以在搜索字符串里给出想匹配的各种标点符号或空白符(空格或制表符等)。换句话说,所给出的搜索字符串必须与想匹配的短语丝毫不差:



```
mysql> SELECT * FROM apothegm
-> WHERE MATCH(attribution,phrase)
-> AGAINST('"bell book and candle"' IN BOOLEAN MODE);
Empty set (0.00 sec)
mysql> SELECT * FROM apothegm
-> WHERE MATCH(attribution,phrase)
-> AGAINST('"bell, book, and candle"' IN BOOLEAN MODE);
+-----+-----+
| attribution          | phrase                      |
+-----+-----+
| Miguel de Cervantes | Bell, book, and candle     |
+-----+-----+
```

- MySQL允许对那些不是FULLTEXT索引组成部分的数据列进行布尔模式的全文本搜索，但这要比那些利用索引而进行的搜索慢很多。

在MySQL 4之前的版本里，改变FULLTEXT搜索参数的办法只有一个——修改源代码并重新编译MySQL服务器。MySQL 4提供了几个可配置的参数，可以通过设置服务器变量的办法来改变它们。我们最感兴趣的两个变量是ft\_min\_word\_len和ft\_max\_word\_len，它们负责确定被索引单词的最小长度和最大长度。这两个变量的默认设置值分别是4和254；在建立FULLTEXT索引的时候，长度不在这个范围内的单词将被忽略掉。

假想想把被搜索单词的最小长度从4改为3，可按以下步骤进行：

- 1) 在启动MySQL服务器的时候把ft\_min\_word\_len变量的值设置为3。为了确保在MySQL服务器启动时能实现这一点，最好把这个设置动作安排到某个选项文件（比如/etc/my.cnf）里去：

```
[mysqld]
set-variable = ft_min_word_len=3
```

- 2) 对于那些已经有FULLTEXT索引的现有数据表，必须重建那些索引。可以先丢弃、再重建那些索引，但下面这种做法更简便：

```
REPAIR TABLE tbl_name USE_FRM;
```

- 3) 在FULLTEXT搜索参数改变后，新创建的FULLTEXT索引将自动使用新的设置值。关于选项文件以及如何设置服务器变量的讨论请参阅附录D。

### 3.10 代码注释

MySQL允许在SQL代码里穿插一些注释，这使我们能够在用来保存数据库查询命令的文件里写上一些说明性的文字。有两种推荐使用的注释样式。第一种，从井字符“#”开始到行尾的所有文字都将被视为注释。第二种，C语言风格的注释也是允许的，即起始标记“/\*”和结束标记“\*/”之间的所有文字都将被视为注释。C语言风格的注释允许跨越多个代码行：

```
# this is a single line comment
/* this is also a single line comment */
/* this, however,
   is a multiple line
   comment
*/
```

第三种注释样式是从MySQL 3.23.3版本开始引入的：注释以两个破折号和一个空格（‘-- ’）开始，从双连字符到行尾的所有文字都将被视为注释。有些数据库软件使用双连字符来开始一条注释；MySQL允许这样做，但要求增加一个空格以避免歧义。否则，诸如“5--7”（5减去-7）之类的表达式就很可能被误认为包含着一条注释。因为不太可能会写出一个“5--7”这样的表达式来，所以MySQL的做法是很实用的。但在实际中，如果所编写的代码不会被移植到仅支持双连字符注释样式的数据库系统里，那还是使用“#”或“/\* ... \*/”形式的注释比较稳妥。

从MySQL 3.22.7版本开始，还可以在C语言风格的注释里“隐藏”一些MySQL独有的关键字，具体做法是：以“/\*!”而不是以“/\*”来作为注释的开头。在遇到这种特殊形式的注释时，MySQL将识别出其中的关键字并执行相应的动作，而其他的数据库服务器却会把它们当做注释的一部分而忽略掉。这种安排增加了代码的可移植性——至少对也支持C语言风格注释的其他服务器来说是这样的：当代码在MySQL环境里执行时，MySQL独有的关键字或者函数能够发挥作用；当需要把代码拿到其他数据库环境里去使用时，它们也用不着修改。下面两条语句在其他数据库服务器看来是等价的，但MySQL在遇到第二条语句时却会执行一个INSERT DELAYED操作：

```
INSERT INTO absence (student_id,date) VALUES(13,'2002-09-28');
INSERT /*! DELAYED */ INTO absence (student_id,date) VALUES(13,'2002-09-28');
```

从MySQL 3.22.26版本开始，还可以在C语言风格的注释里加上版本控制信息：在注释起始标志“/\*!”的后面紧跟着写出一个版本号，那些版本低于3.22.26的MySQL服务器将忽略这条注释中的MySQL独有关键字。比如说，版本号低于3.23.0的MySQL服务器对下面这条CREATE TABLE语句中的注释视而不见：

```
CREATE TABLE t (i INT) /*!132300 TYPE = HEAP */;
```

### 3.11 MySQL不支持的特征

本节将向大家介绍几种可以在其他数据库里找到、但MySQL目前还不支持的功能。但MySQL软件的未来版本对如何实现下面列举的部分功能已经做出了计划。

- **存储过程（stored procedure）和触发器。**存储过程指的是经编译之后被保存在服务器里的一些SQL代码。当MySQL服务器今后需要用到这些代码的时候，就用不着再从客户端把它们发送给服务器去进行分析了。MySQL允许对存储过程做出修改，而这些修改将对会用到该模块的客户端应用程序产生影响。触发器是一种在某个事件发生时（比如删除某个记录的时候）自动调用执行某个存储过程的机制。比如说，如果需要根据数据库中的记录生成某种复杂的汇总统计报告，触发器机制将保证你的报告能够随时反映出数据库里的真实情况。根据最新的开发进度表，存储过程机制将在MySQL 5版本里得到实现。
- **视图。**视图是一种逻辑实体，可以把它当做一个数据表来对待，但它并不是一个真正的数据表。视图提供了一种把来自多个数据表的数据列组织在一起的机制，就好像它们都来自同一个数据表那样。视图有时也被称为“虚拟数据表”（virtual table）。根据最新的开发进度表，视图机制将在MySQL 5版本里得到实现。

- **记录级权限。**MySQL支持多种级别的权限，从全局性权限一直到具体的数据库、数据表和数据列权限。但它目前还不支持记录级权限。不过，可以在自己的应用程序里通过GET\_LOCK()和RELEASE\_LOCK()函数来实现合作式记录锁定，具体做法请参见附录C中的GET\_LOCK()条目。
- **“--”风格的注释。**MySQL之所以不支持这种风格的注释，是因为这种构造很容易导致二义性问题。但从MySQL 3.23.3版本开始，MySQL已经允许使用以“--”（两个破折号加一个空格）开头的注释了。详细情况请参阅第3.10节。

## 第4章 查询优化

关系数据库系统与数学领域中的集合论有着密不可分的关系：数据库是由数据表构成的集合，数据表又是由数据行和数据列构成的集合，而为了从数据表检索出数据行所发出的SELECT查询将使你得到另一个由数据行和数据列构成的集合。在原理性的讨论中，人们经常把数据表里的数据表述为一组与任何一种具体的数据库系统都没有关系的抽象记号，把数据表上的查询操作表述为相应的集合运算。我们知道，数学领域的集合论里是不存在时间概念的，因此，用集合运算来表述查询操作的做法隐含着“查询动作会立刻发生并完成”的意思。

但是，在现实世界里，情况却并非如此。虽然各种数据库管理系统都实现了集合论中的抽象概念，但它们是在各种真实有形的硬件设备上实现的，必须适应各种真实存在着的物理性限制。因此，总是要等一会儿甚至会等很长的时间才能看到查询结果。可人类是一种没有耐心的生物，不喜欢等待。为了实现集合论中瞬间即可完成的数学运算，人们一直在寻找加快查询速度的手段。还好，有些办法的确能让我们做到这一点：人们对数据表进行索引以加快数据库服务器查找数据行的速度；通过各种编程技巧来最大限度地利用那些索引；编写能够影响数据库服务器调度机制的查询以便让来自多个客户程序的查询能够更好地协调工作；修改服务器的操作参数来提高它的工作效率；探究底层硬件的所做所为和绕过各种物理性限制的方法以改善其性能。

以上这些能够加快查询速度的手段就是本章的讨论重点。这一章的目标是帮助大家数据库系统的性能进行优化，使它能够尽可能快地处理查询。MySQL已经相当快了，但对运行查询最快的数据库也可以做这项工作以使它运行得更快。

### 4.1 索引的使用

索引是可以用来加快查询速度的最重要的工具。虽然还可以选用其他的技术，但能否正确地使用索引往往是快、慢之间的最大区别。在MySQL邮件列表上，总有很多人在为了加快查询速度而寻求帮助。令人吃惊的是，因没有使用索引而导致的情况占了相当大的比例，很多问题只要增加一个索引就能得到解决。不过，这一招也不总是那么灵验，因为优化工作并非总是这么简单。但毋庸置疑的是，在没有使用索引的前提下，想用其他招数来大幅改善性能的做法往往收效甚微，纯粹是在浪费时间。要想改善性能，首先应该通过添加索引的办法让性能得到一次最大幅度的提升，然后再考虑其他技术能不能锦上添花。

在这一节里，将对什么是索引以及索引如何改善查询性能等问题进行探讨，还将介绍一些使用了索引反而会适得其反的案例，最后归纳出一套关于如何合理地数据表创建索引的指导原则。在下一节里，将介绍MySQL软件中的查询优化程序。在知道如何去创建各种索引的基础上，再多了解一些与优化程序有关的知识，就能更进一步地用好所创建出来的索引。查询命令的某些具体写法可能会降低索引的使用价值，通常需要避免犯这些错误。（可世事无绝对，有些

场合反而需要去抑制优化程序的行为。我们会在后面的内容里讨论几个这样的例子。)

#### 4.1.1 索引的优点

下面从一个没有任何索引的数据表来开始对索引的研究。一个没有索引的数据表只是一些未经排序的数据行所构成的一个集合。例如,图4-1给出了一个ad数据表,第1章曾介绍过这个数据表。这个数据表没有任何索引,要想找出对应于某个公司的数据行,就必须依次检查数据表里的每一个数据行,看它是否与给定的值相匹配;这就是所谓的“全表扫描”(full table scan)。全表扫描的速度慢、效率低——如果一个很大的数据表里只有几个与筛选条件相匹配的记录,就更难以忍受了。

ad 数据表

company_num	ad_num	hit_fee
14	48	0.01
23	49	0.02
17	52	0.01
13	55	0.03
23	62	0.02
23	63	0.01
23	64	0.02
13	77	0.03
23	99	0.03
14	101	0.01
13	102	0.01
17	119	0.02

图4-1 没有索引的ad数据表

图4-2给出的是同一个数据表,但ad数据表的company\_num数据列上多了一个索引。ad数据表里的每一个数据行在这个索引里都有一个对应的索引项,那些索引项已经按company\_num值排好了序。现在,用不着一个数据行一个数据行地在数据表里查找匹配情况了,我们可以使用索引。假想把与第13号公司有关的数据行都找出来。我们开始扫描那个索引并找到了三个与该公司有关的数据行,然后遇到了一个公司编号为14的数据行,这个编号值大于我们正在查找的值。索引值是已经排好序的,所以遇到公司编号为14的记录就意味着不可能再找到匹配,可以就此退出这次查询了。从这个例子可以看出,索引提高检索效率的原因之一是它可以让我们知道最后一个符合条件的数据行出现在什么位置,此后的数据行都用不着再检查了。索引能够提高检索效率的另一个原因是人们已经发明了很多种定位算法来迅速查出第一个符合条件的数据行出现在什么位置,用不着从索引的开头通过线性扫描去定位一个匹配项(比如说,二分搜索法就要比线性扫描法快很多)。这样,我们就能迅速到达第一个匹配值所在的地方,从而节省了大量的搜索时间。数据库用来定位索引值的技术有很多种,但这里不着重讨论这些技术,重要的是它们可以加快检索速度,而索引也的确是一个加快搜索速度的好方法。

有些读者可能会问:为什么不直接对数据文件进行排序而省掉索引文件?那不是也能让检索速度得到同样的改善吗?是的,能改善——但前提是数据表只有一个索引;如果想再增加一个索引,又怎样才能以两种方式对数据文件进行排序呢?(比如说,也许想建立一个基于顾客姓名的索引和一个基于顾客ID号或电话号码的索引。)把索引创建为独立于数据文件的实体可以



解决这方面的问题并允许人们创建多个索引。此外，索引中的索引项通常都要比数据表中的记录项短一些。当对数据表进行插入或者删除操作时，那些较短的索引值肯定要比那些较长的记录项更容易排序和处理。

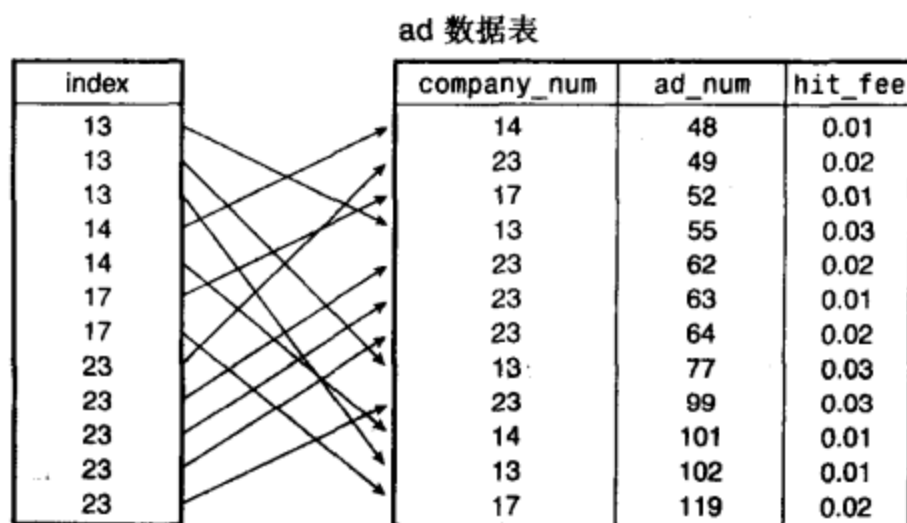


图4-2 增加了索引的ad数据表

这个例子只是对MySQL如何建立索引的原理性描述，不同的数据表类型还有着不同的细节。比如说，对于MyISAM或ISAM数据表，其数据行将被保存在数据文件里，其索引值将被保存在索引文件里。同一个数据表允许有多个索引，这些索引都将保存在同一个索引文件里。索引文件里的一个索引就是一个排好了序的“键-值”型数组，对数据文件的快速访问就是通过这些“键-值”记录而实现的。与此形成对照的是，虽然其中的索引也是一些排了序的数组，但BDB和InnoDB数据表处理程序却都没有把数据行和索引值分开保存。BDB处理程序把同一个BDB数据表的数据值和索引值保存在同一个文件里，InnoDB处理程序则是把所有InnoDB数据表的数据值和索引值都保存在同一个表空间（tablespace）里。

以上介绍的只是索引给单数据表查询带来的好处：索引减少了需要进行全表扫描的次数，因而显著地加快了搜索的速度。然而，索引给涉及多个数据表的关联查询所带来的好处就更大了：在单数据表查询里，最坏情况不过是把所有的数据行遍历一次而已；但在多数据表查询里，最坏情况时的数据行组合个数（各有关数据表中的数据行个数的连乘积）可能是一个天文数字。

假设有三个没有索引的数据表t1、t2和t3，它们分别包含着一个名为c1、c2和c3的数据列，并且各有1000个数据行，数据行的内容都是从1~1000之间的数字。那么，如果想把这三个数据表中取值相等的数据行组合全都检索出来，查询语句可能如下所示：

```
SELECT t1.c1, t2.c2, t3.c3
FROM t1, t2, t3
WHERE t1.c1 = t2.c2 AND t1.c1 = t3.c3;
```

这个查询的结果应该是1000个数据行，每个数据行由三个取值相等的数据列构成。在没有索引的情况下，我们是不可能知道哪个数据行包含着哪个值的。于是，我们就不得不遍历所有的组合才能把与WHERE子句相匹配的数据行找出来。稍有数学知识的人都能计算出可能的组合数将是 $1000 \times 1000 \times 1000$ 个（10亿个！），即最终匹配结果的100万倍。这是一种极大的浪费，即使是在MySQL这种速度已经非常快的数据库上，这类查询的速度也会非常慢。而这还仅仅是

每个数据表只有1000个数据行时的情况，如果这些数据表每个都有几百万个数据行，情况又会怎样呢？如果不使用索引，随着数据行个数的增加，这些数据表上的关联检索所耗费的时间将增长得更快，从而导致非常差的性能表现。如果给每个数据表都增加一个索引，就能大大加快检索的速度，因为索引将使这个查询按以下路线得到处理：

- 从数据表t1中选取第一个数据行，并查出该数据行包含着什么值。
- 利用数据表t2中的索引，直接找到与来自数据表t1的值相匹配的数据行。类似地，利用数据表t3中的索引，再直接找到与来自数据表t1的值相匹配的数据行。
- 前进到数据表t1的下一个数据行，重复上述过程，直到数据表t1里的数据行全都被遍历过一次为止。

此时，我们仍将对数据表t1进行一次全表扫描，但数据表t2和t3中的索引却使我们得以直接提取出这两个数据表中的匹配数据行。从纯数学计算的角度看，此时的查询速度将比不使用索引时的情况快100万倍。（当然，这个例子的目的主要是为了让大家对索引的价值有个比较直接的认识，但它所揭示的问题却是极其现实的。在绝大多数场合，给原本没有索引的数据表加上一个索引往往会使性能得到巨大的改善。）

如上所述，索引不仅能加快MySQL对与WHERE子句相匹配的数据行的检索速度，还能加快关联检索中从其他数据表检索到匹配数据行的速度。此外，索引还能改善其他操作的性能：

- 对于使用了MIN()或是MAX()函数的查询，如果相关的数据列上有索引，MySQL就能直接找到包含着最小值或最大值的数据行，根本用不着去检查每一个数据行。
- MySQL经常使用索引来快速完成ORDER BY和GROUP BY子句所定义的排序和归组操作。
- 有时候，索引甚至能够让MySQL根本不去读取数据行。比如说，假设你打算从某MyISAM数据表的一个数值型数据列（这个数据列上有一个索引）里选取一个值，并且不需要选取这个数据表的其他数据列。此时，MySQL在从索引文件读出索引值的时候就已经查到你要找的值了，根本用不着去读取数据文件——重复读取同一个值既无必要，也没有道理。

#### 4.1.2 索引的缺点

一般说来，如果MySQL能够找出利用索引去加快查询处理速度的方法，它就会这么做。换句话说，在绝大多数场合，如果不对数据表进行索引，受损失的将是你本人。那么，索引是不是只有好处而没有坏处呢？它们真的没有缺点吗？不是，它们也有一些弊端。在实践中，虽然这些弊端几乎总是瑕不掩瑜，但还是应该知道它们到底是什么。

首先，索引需要消耗磁盘空间，索引越多，消耗的空间也就越多。与没有使用索引时的情况相比，你往往会更快地接近甚至超出数据表的尺寸上限：

- 对于ISAM和MyISAM数据表，过多的索引往往会使索引文件先于数据文件达到其尺寸上限。
- 对于BDB数据表，因为它把数据值和索引值都保存在同一个文件里，所以增加索引必然会使它更快地到达BDB数据表文件的尺寸上限。
- InnoDB数据表共同分享着InnoDB表空间里的存储空间，所以增加索引必然会加快InnoDB表空间的消耗速度。不过，只要还能增加磁盘空间（比如增加一个新硬盘），就能通过给InnoDB表空间增加新组件的办法来扩充之。（操作系统的文件尺寸上限对InnoDB表空间没

有约束力，因为InnoDB表空间可以由多个文件构成；这一特性是用来存放ISAM、MyISAM和BDB数据表的文件所不具备的。）

其次，虽然索引能够加快检索操作的速度，但却减慢了被索引数据列上的插入、删除和修改操作的速度。也就是说，索引将减慢大多数需要写入数据的操作的速度。造成这种情况的原因是：在写入一条记录的时候，MySQL不仅要写入一个数据行，而且还必须修改与之有关的所有索引。数据表里的索引越多，需要修改的地方也就越多，对整体性能的负面影响也就越大。在第4.4节里，将对这个问题做进一步的讨论并介绍一些应对措施。

### 4.1.3 挑选索引

索引的创建语法已经在第3.3.4节里介绍过了。可是，知道了索引的创建语法并不代表你掌握了如何对数据进行索引，还需要了解一些使用数据表的方式的知识。本小节内容将帮助大家回答两个问题：应该选择什么样的数据列来创建索引？怎样才能创建出最适用的索引？

- 索引应该创建在搜索、排序、归组等操作所涉及的数据列上，只在输出报告里出现的数据列不是好的候选。换句话说，那些在WHERE子句、关联检索中的FROM子句、ORDER BY或GROUP BY子句中出现过的数据列最适合用来创建索引。只在SELECT关键字后面的输出列清单里出现过的数据列并不是好的候选：

```
SELECT
    col_a                ← 不是好候选
FROM
    tbl1 LEFT JOIN tbl2
    ON tbl1.col_b = tbl2.col_c ← 好候选
WHERE
    col_d = expr;        ← 好候选
```

当然，输出报告中的数据列与用在WHERE子句里的数据列可能是一样的。这一原则的要点是：某个数据列出现在输出列清单里并不是应该对它进行索引的充分理由。

出现在关联检索中的FROM子句或者WHERE子句中的`col1 = col2`表达式里的数据列（比如上例中的`col_b`和`col_c`数据列）都是非常好的索引候选。要是MySQL能把关联检索优化为不需要进行全表扫描的样子，就不必无谓地对相当数量的数据表-数据行组合情况进行检查了。

- 尽量使用惟一化索引。数据值在数据列中的分布情况是一个很值得考虑的因素。建立在惟一化数据列（即数据列中的取值各不相同）上的索引有着最好的效果；如果数据列里有很多彼此重复的值，建立在其上的索引就不会有好的效果。比如说，如果某个数据列里存放着很多各不相同的年龄值，建立在其上的索引就能把不同的数据行很好地区分开；如果某个数据列里存放的是用来表示性别的'M'和'F'两种值，建立在其上的索引恐怕就帮不上多大的忙了——如果数据值的分布比较均匀，那么，不管使用哪个值（'M'或'F'）进行搜索都会匹配到大约50%的数据行。在这种情况下，MySQL可能根本就不会使用建立在这个数据列上的索引——当查询优化程序发现某个数据值在超过30%的数据行里都有出现的时候，它通常会放弃使用相关的索引而进行一次全表扫描。

- **尽量对比较短的值进行索引。**当对一个字符串数据列进行索引的时候，只要有可能，就应该指定一个前缀长度。比如说，假设有一个CHAR(200)数据列，那么，如果大多数数据值的前10个或前20个字节是彼此不同的，就不要对整个数据列进行索引——只对前10个或前20个字节进行索引可以节省大量的空间，而且会使查询进行得更快。比较短的索引需要的磁盘I/O操作比较少，对它们进行比较的速度也更快。更重要的是，键值越短，索引缓存区里容纳的键值也就越多；而MySQL同时保存在内存里的索引项越多，索引缓存区的命中率也就越高。（当然，你也得有点常识。如果只对数据列的第一个字符进行索引，它的用处就可能没有这么大，因为能够用这个索引区分开来的数据值并没有多少。）
- **充分利用最左前缀。**当创建一个由n个数据列构成的复合索引时，实际上是创建了n个索引供MySQL使用。之所以说一个复合索引相当于多个索引，是因为构成这个复合索引的那些数据列的最左集合——即所谓的“最左前缀”（leftmost prefix）——都可以用来匹配数据行。（这与根据某数据列的前n个字节——即该数据列的一个“前缀”——而建立的索引是不一样的。）

假设某个数据表有一个建立在state（州）、city（城市）、zip（邮政编码）数据列上的复合索引，索引项已经按state/city/zip的顺序排好了序，它们也就自动地按state/city和state的顺序排好了序。这样，即使只在查询命令里给出了state值或者state加city值，MySQL也能用上这个索引。也就是说，这个索引可以用于以下几种数据列的组合情况：

```
state, city, zip
state, city
state
```

注意，如果检索操作没有涉及到这个索引的一个最左前缀，MySQL就用不上这个索引了。比如说，如果准备对city或zip进行搜索，这个索引就没有用了。不过，如果你打算搜索某个给定的state和zip值（即构成这个索引的第1个和第3个数据列），那么，虽然这个索引不能用来搜索这两个数据列值的组合，但MySQL仍可使用这个索引来找出匹配的state值，从而缩小了需要进行搜索的范围。

- **适可而止，不要建立过多的索引。**在建立索引的时候，“多多益善”这句话并不适用。试图建立过多的索引反而是一种错误的做法。首先，每增加一个索引，就要消耗一些空间，就会对写操作的性能造成一些负面影响。当对数据表的内容进行修改时，MySQL将不得不为每一个索引做相应的修改，甚至需要重新编排索引里的索引项；索引越多，花费在这方面的时间也就越长。一个很少使用（甚至从未使用过）的索引只会无谓地减慢数据表修改操作的速度。其次，在为查询命令生成执行计划的时候，MySQL必须就每一个索引进行推敲；索引越多，查询优化程序的工作量也就越大。再有，过多的索引有时（这种事并不鲜见）会让MySQL无所适从，反而无法选出最适用的索引。如果只保留必要的索引，就能帮助查询优化程序避免这类错误。

当准备给一个已经有了索引的数据表再增加一个索引的时候，先要想想打算添加的这个索引是不是某个现有的多数据列索引的一个最左前缀。如果是，就没必要再添加它了，因为从使用效果上讲你已经有了一个这样的索引了。（比如说，如果已经有了一个建立在数据



列state、city和zip上的索引，就没必要再增加一个建立在数据列state上的索引了。)

- 考虑将在这个数据列上进行怎样的比较操作。一般说来，索引主要用在<、<=、=、>=、>、BETWEEN等比较操作中。此外，在LIKE操作中，如果匹配模式有一个与某个索引中的键字一模一样的前缀，MySQL也会用到这个索引。可如果某个数据列只会被用在其他类型的操作（比如STRCMP()函数）中，对该数据列进行索引就没有任何价值。此外，HEAP数据表中的索引都是散列化的，并且只会被用在“=”（相等）比较中。也就是说，如果打算对HEAP数据表进行一次与取值范围有关的搜索（比如a < b），它的索引就帮不上你什么忙。
- 利用慢查询日志来找出那些性能低劣的查询。慢查询日志能帮你找出可以利用索引来加快其处理速度的查询来。可以用工具程序mysqldumpslow来查看这个日志。（关于MySQL的各种日志文件的讨论请参阅第11章。）如果经常在慢查询日志里看到某个查询，就应该有“它可能编写得不够优化”的意识。应该试着改写这个查询，看它能不能执行得更快。在查看慢查询日志时，请记住以下几点原则：
  - 这里所说的“慢”是对系统实时性能的一种衡量。与负载较轻的系统相比，在负载较重的系统上，肯定会有更多的查询出现在慢查询日志里。一定要把这个因素考虑进去。
  - 如果在激活慢查询日志功能时还使用了--log-long-format选项，MySQL就会把那些在执行时没有使用索引的查询也记录到这个日志里去，而这些查询不见得都执行得很慢。（尺寸很小的数据表往往不需要建立索引。）

## 4.2 MySQL的查询优化程序

当发出一个用来选取一些数据行的查询命令时，MySQL将对它进行分析，看能不能通过一些优化措施使它执行得更快。在这一节里，将对查询优化程序的工作原理进行介绍。对查询优化程序的详细讨论请参阅MySQL Reference Manual（MySQL使用指南）中与优化工作有关的章节，它对MySQL提供的各种优化手段做了比较深入的描述。

MySQL查询优化程序主要依靠索引来进行查询优化，但它也会使用其他一些信息。比如说，当发出下面这个查询时，不管数据表有多大，MySQL都会非常迅速地执行它：

```
SELECT * FROM tbl_name WHERE 1 = 0;
```

在这个例子里，MySQL只需检查WHERE子句就能知道不存在符合查询条件的数据行，所以它根本用不着去搜索这个数据表。可以用一条EXPLAIN语句来验证这一点，该语句能够让MySQL在没有实际执行SELECT查询的情况下把它的执行路线显示出来。EXPLAIN语句的具体用法很简单，只要把关键字EXPLAIN放在SELECT语句前面就可以了：

```
mysql> EXPLAIN SELECT * FROM tbl_name WHERE 1 = 0;
+-----+
| Comment |
+-----+
| Impossible WHERE |
+-----+
```

通常情况下，EXPLAIN将会返回较多的信息，比如MySQL将使用哪个索引来扫描数据表、



将使用哪种类型的关联操作、需要在各有关数据表里分别扫描多少个数据行的估算值等等。

#### 4.2.1 查询优化程序的工作原理

MySQL查询优化程序有几个工作目标，但它最主要的工作却是尽可能地使用索引，而且是尽可能地使用那些最为“挑剔的”索引以便尽可能多和尽可能快地排除那些不符合查询条件的数据行。这后半句话似乎与我们的目的背道而驰——毕竟，发出SELECT查询的目的是为了得到一些数据行，而不是为了排除它们。查询优化程序之所以会如此工作，是因为把不符合匹配条件的数据行排除得越快，找到符合匹配条件的数据行的速度也就越快。因此，如果率先进行最挑剔的检索测试，查询就能更快地得到完成。假设查询需要对两个数据列进行检验，而这两个数据列上又都有一个索引：

```
SELECT col3 FROM mytable
WHERE col1 = 'some value' AND col2 = 'some other value';
```

我们再假设数据列col1上的测试将匹配到900个数据行，数据列col2上的测试将匹配到300个数据行，而同时满足这两个测试的数据行只有30个。那么，先对col1数据列进行测试的做法将先筛选出900个数据行，然后再从中筛选出与col2值相匹配的30个来，即有870次测试将是失败的；而先对col2数据列进行测试的做法将先筛选出300个数据行，然后再从中筛选出与col1值相匹配的30个来，测试失败的次数减少到了270次。这就大大减少了计算量和磁盘I/O操作。因此，优化程序将先尝试对col2数据列进行测试。

可以依照以下原则帮助优化程序挑选和使用索引：

- **尽量对同类型的数据列进行比较。**当需要对有索引的数据列进行比较的时候，应该尽可能地使用同类型的数据列。比如说，CHAR(10)与CHAR(10)或VARCHAR(10)是同类型的，但与CHAR(12)或VARCHAR(12)则是有区别的。INT与BIGINT也是有区别的。在MySQL 3.23之前的版本里，如果参加比较的数据列不是同类型的，比较操作将不使用这些数据列上的索引。虽然MySQL 3.23及以后的版本对此要求得不那么严格了，但对同类型的数据列进行比较仍要比对不同类型的数据列进行比较有着更好的性能。如果必须对不同类型的数据列进行比较，不妨考虑先用ALTER TABLE语句把其中之一的类型修改为与另一个数据列相同。
- **尽量让有索引的数据列在比较表达式中单独出现。**如果把某个数据列用在了一个函数调用里或者用在了一个复杂的算术表达式里，那么，因为反正要为每一个数据行对表达式进行求值，所以MySQL干脆就不使用这个数据列上的索引了。这种情况有时是无法避免的，但也有许多场合可以改写查询语句以便让有索引的数据列单独出现。

下面的两个WHERE子句就演示了这种情况。它们在数学意义上是等价的，但在优化程序眼里就有高下之分了。在遇到第一个WHERE子句的时候，优化程序将先把表达式4/2简化为数值2，再利用数据列mycol上的索引快速地检索出所有小于2的值来。但在遇到第二个WHERE子句的时候，MySQL必须先检索出每一个数据行里的mycol值，然后乘以2，再把计算结果与4进行比较。此时，因为必须先检索出数据列mycol中的每一个值才能对比较操作符左边的表达式进行求值，所以这个数据列上的索引就派不上用场了：

```
WHERE mycol < 4 / 2
WHERE mycol * 2 < 4
```

我们再来看一个例子。假设有一个带索引的数据列date\_col。当发出一个如下所示的查询时，这个索引将派不上用场：

```
SELECT * FROM mytbl WHERE YEAR(date_col) < 1990;
```

在这个表达式里，与数值1990进行比较的不是那个带索引的date\_col数据列，而是一个根据该数据列计算出来的结果值——MySQL不得不为每一个数据行计算出这个结果值。于是，数据列date\_col上的索引就派不上用场了，这个查询只能通过一次全表扫描才能完成。如何解决这个问题呢？很好办，只要像下面这样使用一个准确的日期值就能利用date\_col数据列上的索引来找出符合条件的数据行了：

```
WHERE date_col < '1990-01-01'
```

可是，当无法给出一个准确的日期值（比如只对从今天算起某给定天数内的记录感兴趣）的时候又该怎么办呢？别着急，这类比较操作也有几种较为优化的表达办法——但它们不见得都同样适用于你的具体情况。有三种可能的表达办法，如下所示：

```
WHERE TO_DAYS(date_col) - TO_DAYS(CURDATE()) < cutoff
WHERE TO_DAYS(date_col) < cutoff + TO_DAYS(CURDATE())
WHERE date_col < DATE_ADD(CURDATE(), INTERVAL cutoff DAY)
```

对于第一个表达式：它无法使用索引，因为MySQL必须先检索出每一个数据行的date\_col值才能计算出TO\_DAYS(date\_col)的值来。第二个表达式要好一些：因为cutoff和TO\_DAYS(CURDATE())都是常数，所以比较操作符右边的表达式将由优化程序在开始处理这个查询之前一次性地计算出来，而不必为每个数据行分别计算一次了。不过，因为date\_col数据列仍出现在一个函数调用里，所以它的索引还是派不上用场。第三个表达式最好：优化程序将在开始处理这个查询之前把比较操作符右边的表达式一次性地计算为一个常数，但这次的计算结果却是一个日期值了。这个日期值能够直接与date\_col值进行比较，不需要再转换为天数。这一回，date\_col数据列的索引终于派上用场了。

- 尽量不要在LIKE模式的开头使用通配符。有时候，人们会用一个如下所示的WHERE子句来对字符串进行搜索：

```
WHERE col_name LIKE '%string%'
```

如果要找到字符串string，不论它出现在col\_name列中的什么位置，那么使用这个表达式就是完全正确的。但千万不要养成在LIKE模式两端随手放上一个“%”字符的习惯。如果想找的只是以string开头的字符串，就应该去掉第一个“%”字符。我们再看一个例子。当需要从一个记录着人们姓氏的数据列里把MacGreGor或MacDougall之类的以“Mac”开头的名字找出来时，应该把WHERE子句写成如下所示的样子：

```
WHERE last_name LIKE 'Mac%'
```

在处理到以纯文本开头的LIKE模式'Mac%'时，优化程序将把这条WHERE子句优化为如下所示的形式，而这种形式将使MySQL利用last\_name数据列上的索引来检索符合条件的数

据行：

```
WHERE last_name >= 'Mac' AND last_name < 'Mad'
```

注意，这种优化手段不适用于使用REGEXP操作符的模式匹配。

- 帮助优化程序对索引的使用效果做出更准确的预测。在默认情况下，当把一个有索引的数据列与一个常数进行比较的时候，优化程序将假设有关索引中的键值是均匀分布的。此外，为了决定是否要使用某个索引来进行常数比较，优化程序还要对这个索引进行一次快速检查以估算可能会用到多少个索引项。对于MyISAM和BDB数据表，可以用ANALYZE TABLE语句让服务器对索引键值的分布情况做一次分析，这将给优化程序提供一些更有价值的信息。另一个办法是执行myisamchk --analyze命令（适用于MyISAM数据表）或者isamchk --analyze命令（适用于ISAM数据表）。注意：因为这两个工具程序都是直接对数据表文件进行操作的，所以用它们来进行键值分析必须满足以下两个条件：
  - 在MySQL服务器主机上，你必须有一个允许对数据表文件进行写操作的账户。
  - 在开始访问数据表文件之前，必须先与服务器搞好协调——在对数据表文件进行分析处理的时候，你肯定不想让别人去访问这个数据表。（用来与服务器协调数据表访问事宜的有关协议请参阅第13章，必须使用正确的写访问协议。）
- 使用EXPLAIN语句来检验优化程序的操作情况。请检查查询所使用的索引是不是能够迅速地排除不符合条件的数据行。如果不是，可以试着使用STRAIGHT\_JOIN强制各有关数据表按指定顺序进行关联。（应该把查询命令执行两次，一次使用STRAIGHT\_JOIN，一次不使用STRAIGHT\_JOIN；然后分析比较两次执行情况的差异。也许MySQL是因为一些更好的理由才没有按你认为最佳的次序来使用那些索引的。）从MySQL 3.23.12版本开始，还可以使用USE INDEX或IGNORE INDEX命令来提示服务器你想让它使用哪些索引。
- 试试查询命令的不同写法，但要多运行它们几次。在试用查询命令的其他写法时，应该把每种形式都多运行几次。如果只把它们分别运行了一次，那经常会发现查询命令的第二种写法执行得更快，但这往往是因为来自前一次查询的信息仍驻留在磁盘缓存区里，因而不需要再从磁盘上读取。另一个需要注意的问题是：为了避免系统上的其他活动对试用结果产生影响，应该尽量在系统负载相对稳定的时候进行测试。
- 不要滥用MySQL的类型自动转换功能。MySQL能够自动进行类型转换，但如果能够避免这类转换，将获得更好的性能。比如说，如果数据列num\_col是一个整数型数据列，下面两个查询将返回同样的结果：

```
SELECT * FROM mytbl WHERE num_col = 4;
SELECT * FROM mytbl WHERE num_col = '4';
```

但第二个查询里有一个类型转换操作。因为需要先把整数和字符串转换为双精度浮点数才能进行比较，所以这个转换操作本身就会对性能产生一个小小的负面影响。更值得注意的是，如果在比较操作中发生了类型转换，有关的索引就将无法派上用场，而这可能对性能造成极严重的影响。

### 4.2.2 抑制优化程序给出的方案

虽然听起来有些奇怪,但有些场合的确需要抑制MySQL查询优化程序给出的方案。下面就是一些需要这样做的理由:

- 想清空一个数据表但希望副作用最小。当需要把一个数据表完全清空的时候,最快速的办法是让服务器先丢弃这个数据表,再根据其.frm文件里的定义把它重新创建出来。TRUNCATE TABLE语句能做到这一点:

```
TRUNCATE TABLE tbl_name;
```

在MySQL 4之前的版本里,用不带WHERE子句的DELETE语句也能获得同样的效果:

```
DELETE FROM tbl_name;
```

在经过优化之后,这种方式的数据表清空操作将执行得非常快,因为MySQL不必逐个地去删除那些数据行了。可是,这种清空操作有以下几个副作用:

- 在MySQL 4之前,哪怕数据表原本不是空白的,不带WHERE子句的DELETE语句也会报告说受影响的数据行个数是零。各MySQL版本里的TRUNCATE TABLE语句也是如此(但会视具体的数据表类型而定)。这会让不了解此事的人觉得疑惑,但并不值得去特别关注。可如果应用程序需要一个“有多少个数据行被删除了”的准确数字,这个0计数值就不可接受了。
- 对于MyISAM数据表,当在其中删除一些数据行的时候,AUTO\_INCREMENT值通常不会被再次使用(请参阅第2章)。但是,如果是用重新创建数据表的方法来清空它的,删除操作就会把AUTO\_INCREMENT序列值重置为从1开始。

如果想避免上述这些副作用,就要使用一个“非优化的”的全数据表DELETE语句,并使该语句的WHERE子句恒为真,如下所示:

```
DELETE FROM tbl_name WHERE 1;
```

这个WHERE子句将使MySQL逐个地删除数据表中的全部数据行,因为它必须为每一个数据行去对WHERE子句进行求值才能决定是否要删掉它。这个查询执行起来当然会慢很多,但它能让你知道到底删除了多少个数据行,并且能把MyISAM数据表当前的AUTO\_INCREMENT序列值保持下来。

- 不想使用优化程序决定的数据表关联顺序。STRAIGHT\_JOIN关键字将迫使优化程序按照给出的顺序对数据表进行关联。在这样做的时候,应该这样来安排各有关数据表在查询命令中的先后顺序:使将从第一个数据表里检索出来的数据行个数最小——(如果无法确定应该把哪一个数据表安排在第一个,就把数据行最多的那个数据表放在第一个。)换句话说,应该尽可能地把最严格的筛选条件安排在最前面。如果能尽早缩小候选数据行的范围,查询命令就会执行得更快。应该把两种关联次序(优化程序决定的和由你指定的)的查询都试试——优化程序不按你认为是最好的顺序去关联那些数据表肯定是有原因的,STRAIGHT\_JOIN关键字不一定能起到加快查询的作用。

另一种做法是这样的:在关联查询命令的FROM子句里,在某数据表的名字后面紧跟着使



用USE INDEX或IGNORE INDEX修饰符，这将使MySQL在检索过程中使用或者不使用该数据表里的索引。如果你认为优化程序做出的决策不正确，可以像这样来帮助它。

- **想按随机顺序来检索查询结果。**从MySQL 3.23.2版本开始，可以用ORDER BY RAND()函数来以随机方式对查询结果进行排序。另一种（适用于MySQL早期版本的）做法是在查询命令里选取一个随机数数据列，然后对查询结果按该数据列进行排序。但要注意的是，如果把查询命令写成如下所示的样子，优化程序是不会按你的意图去进行查询的：

```
SELECT ..., RAND() as rand_col FROM ... ORDER BY rand_col;
```

造成这一问题的原因是：MySQL优化程序发现rand\_col列其实是一个函数调用，于是认为该数据列的取值将是一个常数，进而就把那条ORDER BY子句给优化出了查询！如果想“骗”过优化程序，就要把rand\_col列放到一个表达式里去。比如说，假设数据表里还有一个名为age的数据列，就应该像下面这样写出查询：

```
SELECT ..., age*0+RAND() as rand_col FROM ... ORDER BY rand_col;
```

此时，表达式“age\*0+RAND()”其实就等价于RAND()函数，但优化程序并不明白这一点，于是就不再认为各数据行的rand\_col值是一个恒等的常数值了。

- **想避免数据行修改操作陷入死循环。**在MySQL 3.23.2之前的版本里，当对某个有索引的数据列进行修改的时候，如果该数据列还出现在了WHERE子句里而修改后的索引值又落在了尚未得到处理的范围内，这个修改操作就可能陷入死循环。比如说，假设数据表mytbl有一个带索引的整数型数据列key\_col，下面的查询就会引发这样的问题：

```
UPDATE mytbl SET key_col = key_col+1 WHERE key_col > 0;
```

解决这个问题的办法是：把key\_col数据列放到WHERE子句中的某个表达式里，使MySQL无法使用建立在key\_col数据列上的索引：

```
UPDATE mytbl SET key_col = key_col+1 WHERE key_col+0 > 0;
```

### 4.3 数据列类型与查询效率

选用适当的数据列类型有助于使查询命令执行得更快。在这一节里，我们将就这一问题给出一些指导意见<sup>①</sup>：

- **尽量选用尺寸较小的数据列。**如果使用固定长度的CHAR数据列，就不要让它们超过必要的长度。如果将存放到某个数据列里去的数据的最大长度是40个字节，就不要把它声明为CHAR(255)——把它声明为CHAR(40)已经足够了。同样的道理，如果MEDIUMINT够用，就不要使用BIGINT；这既有助于节约磁盘空间（因为小尺寸的数据表将占用较少的空间），也有助于加快查询速度（因为小尺寸的数据表只需较少的磁盘I/O，而较短的数据值计算起来也更快）。对于那些有索引的数据列，使用较短的数据值还将进一步提升其整体性能——这一方面是因为索引本身就能加快查询的速度，另一方面则是因为短索引值的处理速度要比长索引值的处理速度更快。

<sup>①</sup> 在下面的讨论里，“BLOB类型”应该被理解为BLOB和TEXT两种类型。



- 如果可以选择数据行的存储格式，就应该尽量选用最适用于数据表类型的格式。比如说，如果打算使用MyISAM或ISAM数据表，就应该尽量选用固定长度的数据列而不是可变长度的数据列；对于那些经常需要修改因而容易形成碎片的MyISAM或ISAM数据表就更是如此。具体地说，就是尽量使用CHAR而不是VARCHAR数据列来保存字符串数据。这样做的缺点是数据表将占用较多的空间，但如果你能负担得起额外的空间的话，固定长度的数据行将比可变长度的数据行处理得更快。

对于InnoDB数据表，因为它的数据行内部存储格式对固定长度的数据行和可变长度的数据行不加区分（所有数据行共用一个标头部分，这个标头部分存放着指向各有关数据列的指针），所以使用CHAR类型不见得会比使用VARCHAR类型好。事实上，因为CHAR类型通常要比VARCHAR类型占用更多的空间，所以从减少空间占用量和减少磁盘I/O的角度讲，使用VARCHAR类型反而更有利。

至于BDB数据表，固定长度和可变长度的数据列类型基本上没有什么区别。可以两种存储格式都试试，看哪一种更适用于系统。

- 尽量把数据列声明为NOT NULL。这可以加快处理速度和节约存储空间。此外，因为不再需要把NULL值作为一种特例来检查，所以这往往还能简化查询命令。
- 考虑使用ENUM数据列。如果某个字符串型数据列的不同取值的个数是有限的，可以考虑把它转换为ENUM数据列。ENUM值在MySQL数据库内部被表示为一系列数值，所以它们的处理速度是很快的。
- 利用PROCEDURE ANALYSE()语句。在MySQL 3.23及以后的版本里，可以利用PROCEDURE ANALYSE()语句来分析数据表，看它会对数据列的声明提出哪些建议：

```
SELECT * FROM tbl_name PROCEDURE ANALYSE();
SELECT * FROM tbl_name PROCEDURE ANALYSE(16,256);
```

在PROCEDURE ANALYSE()语句的输出报告里有这样一个输出列，其内容是对数据表里的各数据列的声明情况的优化建议。在上例中的第二条PROCEDURE ANALYSE()语句里，(16,256)（可以根据具体情况改变这两个数字）的含义是：如果某个数据列的不同取值在16个以上或者长度超过了256个字节，就不提出使用ENUM类型的建议。如果不加上这个限制条件，PROCEDURE ANALYSE()语句的输出报告就可能会非常长；ENUM类型的声明通常很难阅读。

根据PROCEDURE ANALYSE()语句的输出报告，就可以把各有关数据列的声明修改为一种更优化的类型了。如果决定改变数据列的类型，可以使用ALTER TABLE语句来进行之。

- 用OPTIMIZE TABLE语句对容易出现碎片的数据表进行整理。数据表——尤其是那些包含有可变长度数据列的数据表——在经过一段时间的使用后，往往会因为数据的修改操作而产生碎片。碎片可不是什么好东西，它会在用来存放那些数据表的硬盘上造成空洞。随着时间的推移，需要读取更多的存储块才能把想要的数据库行读入内存；这无疑会对性能产生不良的影响。包含有可变长度数据列的数据表都会产生碎片，但BLOB数据列——因为它们的长度变化是如此之大——受到的影响往往是最大的。定期使用OPTIMIZE TABLE语句有助于防止数据表查询性能的降低。

OPTIMIZE TABLE语句可以用在MyISAM和BDB数据表上，但只在MyISAM数据表上有碎片整理效果。对各种数据表类型都适用的碎片整理办法是这样的：先用工具程序mysqldump导出数据表，再利用导出文件丢弃并重建之，如下所示：

```
% mysqldump --opt db_name tbl_name > dump.sql
% mysql db_name < dump.sql
```

- 把数据压缩到BLOB数据列里。在应用程序里，用BLOB数据列来保存数据（可以对这些数据进行压缩和解压缩）能让你只用一个检索操作就把所需要的东西都找出来，用不着再使用多个查询命令去查找想要的东西了。BLOB数据列特别适合用来存储那些很难用标准的数据表结构来表示或者会时时变化的数据。在第3章里，当学习ALTER TABLE语句的时候，给出了一个用来充当Web题库的数据表例子。在那个例子里，当需要往题库里增加一道新考题的时候，我们就会使用ALTER TABLE语句给数据表增加一个数据列。

这个问题的另一个解决方案是让负责处理Web表单的应用程序把考题数据压缩到某种数据结构里，然后再把经过压缩的数据插入到一个BLOB数据列里去。比如说，可以把用户给出的考题答案表示为一个XML字符串，再把这个XML字符串保存到一个BLOB数据列里去。这将给应用程序的客户端增加一些负担（对有关数据进行编码和解码），但大大简化了数据表的结构——在需要修改题库的时候，用不着去改变数据表的结构了。

事情总是一分为二的。BLOB值也会给它们自己带来一些麻烦，尤其是在进行了大量的DELETE或UPDATE操作的时候。每一个被删除的BLOB值都会在数据表里留下一个大空洞，这个空洞又将由一些不同长度的记录来陆续填补。（可以用前面介绍的OPTIMIZE TABLE语句来处理这个问题。）

- 人为地给数据表增加一个数据列，而这个数据列的作用就是充当索引。这种“人造索引”数据列往往会有一种出奇制胜的效果。先根据数据表里的其他数据列计算出一个散列值并把它另外保存到一个数据列里，然后通过搜索散列值的办法去检索想找的数据行。注意：这个技巧只适用于精确匹配型查询。（散列值在“<”或“>=”等操作符进行的区间型搜索中毫无用处。）散列值可以用MD5()（适用于MySQL 3.23或更高的版本）、SHA1()（适用于MySQL 4.0.2或更高的版本）、CRC32()（适用于MySQL 4.1或更高的版本）等函数来生成。MySQL 3.23.2之前的版本不允许直接对BLOB或TEXT数据列进行索引，只能在这两种数据列上创建散列索引。但即便是在3.23.2及以后的版本里，通过散列索引去检索BLOB或TEXT值的做法也要比直接检索BLOB或TEXT数据列本身的做法快。
- 尽量避免对大尺寸的BLOB值进行检索——除非万不得已。比如说，如果不能断定WHERE子句将把查询结果限制为想要找的哪些数据行，贸然使用一个SELECT \*查询的做法就不可取——很可能会毫无必要地让数据库服务器把一些尺寸非常大的BLOB值通过网络传输给你。这类场合是刚刚介绍的“人造索引”（即BLOB值的散列标识信息）大显身手的又一个领域：你应该先搜索那个“人造索引”数据列以确定哪些数据行符合筛选条件，然后再把符合条件的BLOB值从服务器检索到客户端来。
- 把BLOB值剥离到另一个数据表里去。在某些场合，把数据表里的BLOB数据列剥离出来并存放到另一个数据表——如果这有助于把数据表里的其他数据列转变为固定长度的数据

行格式的话——往往是一个不错的主意。这既能减少原始数据表里的碎片，使你享受到使用固定长度的数据行所带来的好处；又能使原始数据表上的SELECT \*查询不会把大尺寸的BLOB值不必要地通过网络传输给你。

#### 4.4 更有效地加载数据

因为SELECT查询最为常见，而它们当中又有一些是不那么容易优化的，所以在说到优化的时候，人们的注意力大都集中在SELECT查询上面。把数据加载到数据库里的操作则相对要简单明了得多。话虽如此，仍有一些策略有助于改进数据加载操作的效率。下面就是一些基本原则：

- 与一个一个地去加载数据行的做法相比，批量加载的效率更高——因为MySQL不必每加载一条记录就刷新一次索引，它可以等记录全部加载完毕后再刷新索引。而索引的刷新次数越少，数据加载得也就越快。
- 与有索引时的情况相比，数据表没有索引时的数据加载操作完成得更快。对于有索引的数据表，不仅要把新记录本身添加到数据文件里，而且还必须对各个索引做出相应的修改。
- 短SQL语句比长SQL语句执行得更快——因为前者在服务器端的词法分析工作量较少，经网络从客户端发送到服务器端所需要的时间也较短。

这些因素虽然微不足道（尤其是最后一个因素），但在需要加载大量数据的场合却会导致相当巨大的差别。根据这些基本原则，可以得出以下几个有助于加快数据加载操作的结论：

- LOAD DATA（各种格式）语句要比INSERT语句的效率高。LOAD DATA语句以批量方式来加载数据行，索引的刷新操作发生得不频繁，需要服务器去分析和执行的语句也只有一条（而不是好几条）。
- LOAD DATA语句要比LOAD DATA LOCAL语句的效率高。LOAD DATA语句要求有关文件存放在服务器上且你必须有相应的FILE权限，但服务器可以直接从本地磁盘读到文件内容。LOAD DATA LOCAL语句则需要由客户（程序）去读取文件并把其内容通过网络送往服务器，这就要慢一些了。
- 如果必须使用INSERT语句，请尽量像下面这样用一条语句去插入多个数据行：

```
INSERT INTO tbl_name VALUES(...),(...),... ;
```

在一条语句里给出的数据行越多，效果越好。这既能让你少发出几条INSERT语句，又能减少索引被刷新的次数。这与“短SQL语句比长SQL语句执行得更快”的原则看似矛盾，其实不然：与一组等价的单数据行INSERT语句相比，一次插入多个数据行的INSERT语句还是要短一些；后者让服务器刷新索引的次数也较少。

在用工具程序mysqldump生成数据库备份文件的时候，应该尽量使用--extended-insert选项，这将使导出文件的内容是一些可以一次插入多个数据行的INSERT语句。也可以使用--opt（意思是“optimize”，优化）选项，它将自动启用--extended-insert和其他几个选项，而这些选项将使导出文件在被重新加载的时候能够得到更有效率的处理。同样的道理，应该尽量避免使用mysqldump程序的--complete-insert选项——这个选项生成的INSERT语句每条只能插入一个数据行，这样将增加导出文件的长度和数据加载工作中的词法分析工作量。

- 如果必须使用多条INSERT语句，请尽量把它们集中在一起以减少对索引的刷新次数。对于支持事务处理机制的数据表类型，应该把这些INSERT语句放在同一个事务里而不是在自动提交模式下执行它们，如下所示：

```
BEGIN;
INSERT INTO tbl_name ... ;
INSERT INTO tbl_name ... ;
INSERT INTO tbl_name ... ;
COMMIT;
```

对于不支持事务处理机制的数据表类型，应该先对数据表进行写锁定，然后在数据表锁定期间发出这些INSERT语句，如下所示：

```
LOCK TABLES tbl_name WRITE;
INSERT INTO tbl_name ... ;
INSERT INTO tbl_name ... ;
INSERT INTO tbl_name ... ;
UNLOCK TABLES;
```

这两种做法都将使你获得同样的益处：索引将只刷新一次，而不是像在自动提交模式下或者事先没有锁定数据表时那样每执行一条INSERT语句就刷新一次。

- 利用客户/服务器通信协议中的压缩功能来减少网络上的数据传输量。对大多数MySQL客户程序来说，在命令行上给出--compress选项将能做到这一点。不过，因为压缩工作需要消耗大量的资源，所以通常只适用于慢速的网络环境。
- 让MySQL插入默认值。也就是说，不要在INSERT语句里把那些将被赋予其默认值的数据列写出来。语句的平均长度越短，需要经网络送往服务器的字符个数也就越少；语句中包含的数据值越少，服务器端的词法分析和类型转换工作量也就越少。
- 对于有索引的数据表，可以通过批量插入数据行（使用LOAD DATA语句或者多数据行INSERT语句）的办法来减少因需要对索引进行刷新而导致的开销。因为MySQL只需在有关数据行都加载完毕后对索引进行一次刷新而不是每插入一个数据行就刷新一次索引，所以索引刷新方面的工作量将被减少到最低水平。
- 对于MyISAM和ISAM数据表，如果需要把大量数据加载到一个新数据表里去，就应该先创建一个没有索引的数据表、再加载数据、然后再创建索引。“先加载数据、后创建索引”要比“先创建索引、后加载数据”的做法更有效，因为MySQL不必每加载一个数据行就去刷新一次索引。如果数据表已经有了索引，“事前先丢弃或禁用之，事后再重建或激活之”的做法通常会使数据加载工作更有效。注意，这两种方法不适用于InnoDB或BDB数据表，即使这样做了，也不会有优化效果。

在考虑使用丢弃或者禁用索引的策略来为MyISAM或ISAM数据表加载数据之前，应该先根据具体情况对这样做是否有利做一个全面的分析。比如说，如果只需要把很少的数据加载到一个很大的数据表里去，重建索引策略就可能还不如老老实实地依次插入各数据行的办法快。

索引的丢弃和重建可以用DROP INDEX和CREATE INDEX语句或者相应的ALTER TABLE语句来完成。禁用和重新激活索引的办法则有两种：



- 使用ALTER TABLE语句的DISABLE KEYS和ENABLE KEYS，如下所示：

```
ALTER TABLE tbl_name DISABLE KEYS;
ALTER TABLE tbl_name ENABLE KEYS;
```

这两条语句用来禁止或者允许对数据表非惟一化索引的刷新操作。

- 使用myisamchk或isamchk工具程序。这两个程序能对索引进行操作，但它们将直接在数据表文件上进行操作，只有具备了数据表文件的写权限才能使用它们。此外，为了防止数据库服务器在你正使用着数据表文件的时候对之进行访问，应该按第13章中给出的有关步骤进行操作。

DISABLE KEYS和ENABLE KEYS语句是用来禁用和重新激活索引的首选手段，因为有关工作是在服务器端完成的。不过，这两个语句出现于MySQL 4及以后的版本里。（注意，如果是用LOAD DATA语句来把数据加载到一个空白MyISAM数据表的，服务器将自动进行这种优化。）

要想“手动地”禁用某MyISAM数据表的索引，先要让数据库服务器停止使用该数据表，然后进入该数据表所在的数据库目录并发出如下所示的命令：

```
% myisamchk --keys-used=0 tbl_name
```

把数据加载到数据表里以后，再使用以下语句重新激活索引：

```
% myisamchk --recover --quick --keys-used=n tbl_name
```

$n$ 是用来表明需要激活哪些索引的位掩码，第0位对应着第一个索引。比如说，如果数据表有三个索引， $n$ 值就应该是7（即二进制的111）。索引的编号可以用--description选项来确定：

```
% myisamchk --description tbl_name
```

对ISAM数据表进行操作的命令与此类似，只是要使用isamchk而不是myisamchk程序。另外，isamchk程序的--keys-used值代表的是数据表中的索引的最高编号。（如果数据表有三个索引， $n$ 值就应该是3。）

上面介绍的这些数据加载操作原则也适用于同时有多个客户程序在进行不同类型的查询的情况。比如说，应该尽量避免在修改（写入）操作比较频繁的数据表上使用那些需要很长时间才能运行完毕的SELECT查询，因为它们会导致各种竞争现象并使得进行写操作的客户程序性能下降。如果写操作大都是INSERT操作，不妨采用下面这个办法来绕道而行：先把记录添加到一个辅助性的数据表里，然后再定期地把记录添加到主数据表里去。当然，如果需要立刻访问新添加的记录，这个办法是不太合适的。可如果可以短时间内不访问它们，增加一个辅助性的数据表将带来两个好处：首先，正式数据表上的SELECT查询所导致的竞争现象将大为减少，从而加快了它们的执行速度；其次，与一条一条地把记录直接加载到主数据表里的做法相比，把记录从辅助数据表批量加载到主数据表的做法将花费更少的时间；索引只需在批量加载操作完成时进行一次刷新而不是每加载一条记录就刷新一次。

这是个很实用的策略。比如说，当你想把来自Web服务器的各个Web页面保存到一个MySQL数据库里去时（此时，能否在修改某个Web页面后立刻把它放到主数据表里并不是件非常紧迫的事），就可以这样做。

对于MyISAM数据表，如果打算加载的数据没有重要到必须在系统异常关闭之前一个不少



地全都被插入到数据表里去（比如在用MySQL数据库来保存某种日志而该日志却不是十分重要的场合），在创建数据表时使用DELAYED\_KEY\_WRITE选项也是一个能减少索引刷新次数的策略——这将使MySQL在其不那么繁忙时才去刷新索引缓存区而不是在每插入一条记录就立刻刷新之。如果想在数据库服务器上全面推行这种“推迟索引刷新操作”的策略，可以用--delay-key-write选项来启动mysqld程序。此时，数据表的索引块只有在它必须被刷新（比如，需要为其他的索引值腾出地方、执行了一条数据表刷新命令或者数据表被关闭）时才会被真正地刷新。

在镜像机制中的从服务器上，可以用--delay-key-write=ALL选项来强行延迟所有MyISAM数据表（不管它们当初在主服务器上是如何被创建的）上的索引刷新操作。

## 4.5 调度和锁定问题

前面几节的讨论主要集中在怎样才能让一个单个的查询执行得更快。MySQL还允许改变各种语句的调度优先级，这将有助于使来自多个客户程序的查询合作得更好，使每一个客户程序都不会阻塞得太久。改变优先级还能使某些特定类型的查询得到更快的处理。在这一节里，将介绍MySQL的默认调度策略、可以用来影响这一策略的相关选项、数据表处理程序的锁定机制和级别对并发运行中的客户程序的影响等等。在下面的讨论中，将把进行检索（SELECT）操作的客户程序称为“读者”（reader），把对数据表进行数据修改（DELETE、INSERT、REPLACE或UPDATE）操作的客户程序称为“写者”（writer）。

MySQL的基本调度策略可以归纳为以下两条：

- 写入请求将按它们到达服务器的先后顺序得到处理。
- 写操作的优先级要高于读操作。

MyISAM和ISAM数据表的调度策略是在数据表锁定的帮助下实现的。在客户程序访问数据表之前，必须先获得相应的（操作）锁；只有在完成了对数据表的操作之后，这把锁才能被释放。可以明确地通过发出LOCK TABLES或UNLOCK TABLE语句来申请或者释放有关的锁，但这项工作通常可以交给服务器去代劳——服务器的操作锁管理组件能自动地在需要时去申请操作锁，等用完之后再自动地释放之。

进行写操作的客户程序必须先申请到允许其独占性地访问有关数据表的锁。在写操作的进行过程中，数据表处于一种不稳定的状态——有关记录正在被删除、添加或者修改，而数据表上的索引也需要做相应的刷新。在此期间，如果还允许其他客户程序访问这个数据表，就可能导致各种问题。很明显，允许两个客户程序同时对同一个数据表进行写操作是件很不好的事，因为很容易把数据表弄得一团糟。类似地，允许客户程序在数据表尚在变化时就对它进行读操作也同样是不好的，因为如果修改操作和读操作恰好发生在同一个地点，读到的结果就将是 inaccurate 的。

进行读操作的客户程序必须先申请不允许其他客户程序对数据表进行写操作的锁，以此来确保数据表不会在读操作进行过程中发生变化。不过，读操作锁并不需要是独占性的，它允许其他客户程序同时从数据表上读取数据。读操作不会改变数据表里的数据，所以“读者”也不必彼此排斥。

MySQL允许通过几个查询修饰符来影响其调度策略，比如LOW\_PRIORITY（适用于DELETE、INSERT、LOAD DATA、REPLACE、UPDATE等语句）、HIGH\_PRIORITY（适用于SELECT语句）、DELAYED（适用于INSERT和REPLACE语句）等关键字。

LOW\_PRIORITY关键字对调度机制的影响是这样的：在正常情况下，如果数据表在一个写操作到达时正在被读取，这个“写者”就将阻塞到“读者”完成为止。换句话说，查询一旦开始，就不能中断。如果在“写者”等待期间又到来了一个“读者”，这个“读者”也将阻塞，因为默认的调度策略是“写者”比“读者”的优先级高。当第一个“读者”完成之后，“写者”开始工作，“写者”完成之后，第二个“读者”才能开始工作。

可如果这个写请求是一个LOW\_PRIORITY（意思是“低优先级”）请求，这个“写者”的优先级就比“读者”低。此时，如果在“写者”正在等待期间又来到了第二个读请求，第二个“读者”将被允许加塞到“写者”的前面；只有当没有“读者”在排队等候的时候，“写者”才能开始工作。从理论上讲，这种调度机制存在着使LOW\_PRIORITY写请求被永久阻塞的可能性——如果在前一个“读者”尚未完成时又有新的读请求不断地到来，新到的读请求就总是会加塞到LOW\_PRIORITY写请求的前面。

SELECT查询中的HIGH\_PRIORITY关键字有着类似的作用：虽然“写者”的优先级在正常情况下要高于“读者”，但HIGH\_PRIORITY关键字将使SELECT查询加塞到一个正在等待中的“写者”之前。

INSERT语句中的DELAYED修饰符的工作情况是这样的：当某数据表上的一个INSERT DELAYED请求到达时，服务器将把随之而来的数据行放入一个“延迟插入”队列，然后向客户程序返回一个状态信息，使客户程序能够在新数据行还没有被实际插入到数据表里之前继续往下执行。如果刚好有“读者”正在读取那个数据表，新数据行就会一直待在队列里。一直等到没有“读者”的时候，服务器才会开始把“延迟插入”队列里的数据行真正地插入到相应的数据表里去。在进行延迟插入的过程中，服务器会时不时地检查是否有新的读请求到来并正在等待处理。如果有，MySQL将挂起“延迟插入”队列，让“读者”开始运行。等把“读者”都处理完以后，服务器将再次开始进行延迟插入。这一过程将持续到“延迟插入”队列被清空为止。

LOW\_PRIORITY和DELAYED修饰符的共同点是它们都将使数据行的插入动作被推迟进行，但它们对客户程序的影响是不一样的。LOW\_PRIORITY迫使客户程序一直等到可以插入数据行为止；DELAYED允许客户程序继续往下执行——新数据行将由服务器缓存起来，等空闲时再进行插入。

INSERT DELAYED特别适用于这样的场合：其他客户程序正运行着一个冗长的SELECT语句，而你又不想被阻塞并不得等到插入操作完成之后才能继续自己的工作。发出INSERT DELAYED语句的客户程序几乎无需等待就能继续往下执行，因为服务器只需简单地把将被插入的数据行放到“延迟插入”队列里就行了。

注意，普通的INSERT语句与INSERT DELAYED语句是有区别的。对于普通的INSERT语句，服务器会向客户程序提供较多的返回信息；对于INSERT DELAYED语句，服务器只在该语句有语法错误时返回一条出错信息——如果语句没有语法错误，就连这条信息也不返回。比如说，在使用INSERT DELAYED语句的时候，将无法在该语句返回时获得AUTO\_INCREMENT值，也

无法知道这次插入会在惟一化索引里导致多少个重复的索引项。之所以会这样，是因为INSERT DELAYED的执行状态信息是在插入操作真正完成之前返回的。此外，当用INSERT DELAYED语句插入的数据行仍在队列中等着被插入数据表时，如果服务器崩溃了或者用kill -9命令中止了服务器进程，那些数据行就丢失了。但普通的kill -TERM命令不会如此“无情”——服务器会在退出执行之前把那些数据行都插入到有关的数据表里。

“‘读者’阻塞‘写者’”的原则在MyISAM数据表上有一个特例。这个特例发生在MyISAM数据表里没有任何空洞（即从未在这个数据表里删除过数据行或者刚刚对它进行过碎片整理）的情况下，此时，用INSERT语句插入的数据行只会被追加到数据表的末尾，不可能出现在中间位置。对于这样的MyISAM数据表，MySQL允许在有其他客户正在读取数据表的同时向其中添加数据行。人们把这种插入操作称为“并发插入”（concurrent insert），因为它们能够与检索操作同时进行而不会被阻塞。如果想使用这个技巧，就要注意以下几个问题：

- 不要在INSERT语句中使用LOW\_PRIORITY修饰符。它会使INSERT语句总是阻塞“读者”，并发插入也就无从谈起了。
- 如果某个“读者”既要明确地对数据表进行锁定，又想给并发插入留个“后门”，就必须使用LOCK TABLES ... READ LOCAL而不是LOCK TABLES ... READ语句来进行锁定。用关键字LOCAL申请到的操作锁将允许进行并发插入——它只对数据表中已经存在的数据行进行锁定，不会阻塞把新数据行追加到数据表末尾的插入操作。

能影响MySQL调度机制的修饰符并非同时出现在MySQL里。下表列出了允许使用这类修饰符的语句以及它们始见于MySQL软件的哪个版本。可以根据这个表来判断你的服务器都支持哪些用法。

语句类型	最初出现的版本号
DELETE LOW_PRIORITY	3.22.5
INSERT LOW_PRIORITY	3.22.5
INSERT DELAYED	3.22.15
LOAD DATA LOW_PRIORITY	3.23.0
LOAD DATA CONCURRENT	3.23.38
LOCK TABLES ... LOW_PRIORITY WRITE	3.22.8
LOCK TABLES ... READ LOCAL	3.23.11
REPLACE LOW_PRIORITY	3.22.5
REPLACE DELAYED	3.22.15
SELECT ... HIGH_PRIORITY	3.22.9
UPDATE LOW_PRIORITY	3.22.5
SET SQL_LOW_PRIORITY_UPDATES	3.22.5

## 锁定级别与并发问题

上面介绍的调度修饰符使我们能够影响MySQL的默认调度策略。它们中的大多数是人们为了解决数据表级锁定——MyISAM和ISAM处理程序就是使用这种锁定机制来处理数据表竞争问题的——一时可能发生的各种问题而引入的。

如今, MySQL又增加了BDB和InnoDB数据表处理程序, 它们能够在多种不同的级别对有关的数据表实施锁定, 而这些机制在数据表竞争管理方面有着不同的特性。BDB处理程序使用页面级操作锁。InnoDB处理程序使用数据行级操作锁, 但只在必要时才这样做。(在很多场合, 比如只需进行读操作的时候, InnoDB处理程序可能根本就不使用操作锁。)

数据表处理程序所使用的锁定级别对客户程序的并发运行有着显著的影响。例如, 假设有两个客户程序都要修改某给定数据表里的某一个数据行。为了进行修改, 这两个客户程序都必须先申请一个写操作锁。对于MyISAM数据表, 处理程序会让第一个客户程序锁定数据表, 第二个客户程序则会被阻塞到第一个客户程序完成操作为止; 两个客户程序不会并发运行。对于BDB数据表, 只要这两个客户程序想要修改的数据行不在同一个页面上, 两个修改操作就可以同时进行; 两个客户程序有可能(可能性很大)并发运行。对于InnoDB数据表, 只要修改的不是同一个数据行, 两个修改操作就能同时进行; 换句话说, InnoDB数据表对并发机制的支持程度最高。

从原则上讲, 锁定级别越细小, 对并发机制的支持就越好, 也就允许有更多的客户程序在同一时间去使用同一个数据表的不同部分。从实践上讲, 不同的数据表类型适用于不同的查询混用局面:

- ISAM和MyISAM数据表的检索速度非常快。但在检索和修改操作相混杂的情况下, 使用数据表级操作锁就会成问题, 尤其是在检索操作需要花费很长时间的时候。在这种局面里, 修改操作很可能会等待很长时间才能得到处理。
- BDB和InnoDB数据表能够在有大量修改操作的时候提供更好的性能。因为锁定发生在页面级或者数据行级, 数据表的被锁定范围也就比较小。这就减少了对操作锁的竞争, 改善了并发性。

从预防死锁现象的角度看, 锁定级别越细小, 效果也就越好。如果你使用的是数据表级的操作锁, 死锁现象根本不会发生——服务器将从查询命令中分析出本次查询都会用到哪些数据表并在开始查询之前把它们全都锁定。BDB和InnoDB数据表则存在着发生死锁现象的可能性——BDB或InnoDB处理程序不会在事务开始之初把所有必要的操作锁都申请到, 它们将在事务处理过程的必要阶段才去申请必要的操作锁。这就有可能导致这样的局面: 两个已经各自申请到一把操作锁的查询在某一时刻发现自己需要再申请一把操作锁, 但它们申请到新操作锁的前提却是对方释放掉现有的操作锁。于是, 这两个客户程序都拿着对方继续执行所必需的操作锁, 却又必须等待对方释放掉其操作锁之后才能继续执行。死锁现象就这样发生了, 而服务器则必须放弃这两个事务中的一个。对于BDB数据表, 用LOCK TABLES语句来明确地申请一把数据表级别的操作锁有助于预防死锁现象的发生——因为BDB处理程序能够识别出数据表级的操作锁并做出相应的处理。这个办法不适用于InnoDB数据表, 因为其处理程序识别不出用LOCK TABLES语句申请的操作锁。

## 4.6 系统管理员所完成的优化

前面几节介绍的优化措施是非特权MySQL用户都能进行的, 还有一些优化措施只能由负责管理MySQL服务器(程序或主机)的系统管理员来进行。在各种允许进行细调的服务器参数当



中，有些与MySQL的查询处理工作有着密切的关系，而硬件配置方面的某些问题对查询处理速度的快慢也有着直接的影响。概括地讲，当以系统管理员的身份优化MySQL时，应该记住以下几个基本原则：

- 内存里的数据要比磁盘上的数据访问起来更快。
- 让数据尽可能长时间地留在内存里能减少磁盘读写活动的工作量。
- 让索引信息留在内存里要比让数据记录的内容留在内存里更重要。

下面是上述原则的几个具体实施办法：

**增加服务器的缓存区容量。**服务器有许多允许人们加以改变的参数（变量），这些参数的改变又将影响到它的工作状况。在这些服务器参数中，有几个对查询处理速度有着直接的影响。在允许人们加以改变的服务器参数中，最重要的就是数据表缓存区和数据表处理程序用来完成各种索引操作的缓存区容量了。如果有足够的内存，就应该把MySQL服务器的各种缓存区和缓冲区的容量尽可能地设置得大一些，这有助于延长信息驻留在内存里的时间，减少磁盘读写活动。因为访问内存里的信息要比去读写磁盘上的信息快，所以这是一种很好的优化措施。

- 数据表缓存区里存放着与打开的数据表有关的各种信息，它的容量由服务器变量 `table_cache` 控制。如果服务器已经打开了很多数据表，这个缓存区就可能会被填满——如果服务器需要再打开一些数据表；就必须先把那些使用率不高的数据表关掉以腾出地方。如果想知道数据表缓存区的使用效率，可查看状态变量 `Opened_tables`，如下所示：

```
SHOW STATUS LIKE 'Opened_tables';
```

当MySQL服务器需要用到某个数据表而该数据表从未被打开过或已经被关闭时，服务器就得先打开它才能进行后续操作；每进行一次数据表打开操作，服务器就会给 `Opened_tables` 值——服务器进行过多少次数据表打开操作的总次数——加上一个1。（`mysqladmin status` 命令的输出报告将把这个数字显示为 `Opens` 值。）如果这个数字保持稳定或稳步增加，就说明数据表缓存区的容量设置得比较合理。可如果这个数字增加得很快，就意味着数据表缓存区已满，服务器必须先关闭一些数据表才能为打开其他数据表而腾出地方。如果有足够的文件描述符可用，增加数据表缓存区的容量将减少数据表打开操作的次数。

- 为了进行与索引有关的操作，MyISAM和ISAM数据表处理程序需要把索引块缓存在键字缓冲区（`key buffer`）里。这个缓冲区的容量由服务器变量 `key_buffer_size` 控制。容量越大，能被缓存在内存里的索引块也就越多，MySQL不需要从磁盘读入新索引块就能找到某给定键值的可能性也就越大。键字缓冲区的默认容量是8MB，这对内存充足的系统来说是相当保守的。在条件允许的情况下，增加容量将给基于索引的检索操作以及索引的创建和修改操作带来显著的性能改善。
- InnoDB和BDB数据表处理程序各有一个供自己专用的数据与索引值缓存区，它们的容量分别由服务器变量 `innodb_buffer_pool_size` 和 `bdb_cache_size` 控制。InnoDB处理程序还维护着一个日志缓冲区，其容量由服务器变量 `innodb_log_buffer_size` 控制。
- 还有一种特殊的缓存区叫做查询缓存区，将在第4.6.1节里对它做专门的介绍。

服务器变量的设置办法参见第11章。如果想修改某个参数的值，请遵循下列原则：



- 每次只改变一个参数。同时改变多个不相关联的参数值会使其效果难以评估。
- 循序渐进地调整服务器变量的值。教条地依据“多多益善”的理论分析而大幅增加服务器变量值的做法会迅速耗尽系统上的资源，过高的设置值反而会使系统濒于崩溃或者慢如蜗牛。
- 要想根据系统的具体情况对适当的服务器变量做出适当的设置，请仔细查阅MySQL发行版本中自带的my-small.cnf、my-medium.cnf、my-large.cnf、my-huge.cnf等选项文件。（这几个文件通常存放在源代码发行版本的support\_files目录或者二进制发行版本的share目录里。）这些文件能让你了解不同规模的MySQL服务器各有哪些参数最值得修改以及这些参数又有哪些典型的设置值。

下面是能帮助服务器运行得更有效的其他一些策略：

- **禁用用不着的数据表处理程序。**服务器不会为被禁用的处理程序分配任何内存，节省下来的内存可以用到其他更需要的地方。如果MySQL服务器是从源代码开始建立的，就可以彻底禁用ISAM、InnoDB和BDB处理程序；InnoDB和BDB处理程序还可以在服务器启动时加以禁用。详细情况参见第11章。
- **权限表里的权限关系要尽可能地保持简单。**虽然服务器会把权限表（grant table）的内容缓存在内存里，但如果tables\_priv或columns\_priv数据表里有一些数据行，服务器就将不得不为每一条查询命令去核查相应的数据表级权限和数据列级权限。如果两个数据表都是空的，服务器在对其权限检查工作进行优化的时候就能省略掉这两个核查步骤了。
- **在从源代码开始建立MySQL的时候，尽量使用静态库而不是共享库来完成其配置工作。**使用了共享库的动态二进制代码可以节约磁盘空间，但静态二进制代码的执行速度更快。可如果还打算加载用户定义函数（user-defined function，简称UDF）的话，就不能使用静态库了——UDF机制必须依赖动态链接才能实现。

#### 4.6.1 查询缓存区

在实际工作中，有些SELECT语句经常会被反复多次地执行。从MySQL 4.0.1开始，服务器可以使用查询缓存区（query cache）来加快对这类SELECT语句的处理，因此而得到的性能改善往往相当可观。查询缓存区的工作情况是这样的：

- 当第一次执行某给定SELECT语句的时候，服务器会记住这条查询命令的文本和它的返回结果。
- 等再次遇到这个查询的时候，服务器不会浪费时间去再次执行之。相反，服务器会直接从查询缓存区里把相应的查询结果提取出来并返回给客户程序。
- 查询缓存机制是以服务器接收到的查询命令字符串是否一模一样为判断依据的。如果前、后两次查询的命令文本一模一样，MySQL就认为它们是相同的。如果查询命令在字母的大小写方面有差异，或者来自使用着不同的字符集或通信协议的客户程序，MySQL就会认为它们是不同的。此外，即使前、后两次查询的命令文本一模一样，可如果它们指称的数据表并非同一个（比如两个名字完全相同却分别来自不同数据库的数据表），MySQL也将把它们视为不同的查询。

- 如果某个数据表被刷新了，查询缓存区里与之有关的一切查询就都将失效并被丢弃。这是为了避免服务器把已经过时的查询结果返回给客户程序。

对查询缓存区的支持已默认地内建在MySQL服务器里了。如果不想使用这个缓存区，也不想因为它而额外增加哪怕最小的开销，可以利用configure脚本的--without-query-cache选项来建立一个不带查询缓存机制的服务器。

对于那些支持查询缓存机制的服务器，缓存操作将由以下三个服务器变量控制：

- `query_cache_size`：查询缓存区的尺寸。如果这个变量值是0，则表明禁用查询缓存机制；这是它的默认值。（换句话说，如果想使用这个缓存区，就必须在事先明确地启用之。）启用这个缓存区的办法很简单，给`query_cache_size`变量设置一个适当的值（以字节为计量单位）就行了。比如说，在选项文件里，可以像下面这样把查询缓存区的尺寸设置为16MB：

```
[mysqld]
set-variable = query_cache_size=16M
```

- `query_cache_limit`：将被缓存的结果集的最大尺寸（以字节为计量单位），如果某个查询的结果集总长度大于这个数字，MySQL就不会把它缓存到查询缓存区里。
- `query_cache_type`：查询缓存区的操作模式。查询缓存区的操作模式有以下三种：

模 式	含 义
0	不进行缓存
1	除以SELECT SQL_NO_CACHE开头的查询以外，对其他查询都进行缓存
2	“按要求缓存”，即只对以SELECT SQL_CACHE开头的查询进行缓存

在默认的情况下，服务器将根据自己当前的查询缓存区操作模式对来自各客户程序的查询进行处理，但客户程序可以通过发出下面这条语句来改变服务器对来自该客户程序的查询的缓存行为：

```
SET SQL_QUERY_CACHE_TYPE = val;
```

`val`的可取值有0、1、2三种，其含义与`query_cache_type`变量值相同；而符号值OFF、ON、DEMAND分别是0、1、2的同义词。

当你在客户程序里发出一条查询命令的时候，还可以通过这条查询命令本身向服务器表明你是否想让它把这次查询的结果缓存到查询缓存区里去——只需在SELECT关键字的后面加上一个修饰符就可以做到这一点：SELECT SQL\_CACHE表明你想让服务器把本次查询的结果缓存起来（如果查询缓存区的操作模式是1或2，这个查询就将被缓存）；SELECT SQL\_NO\_CACHE表明你不想让服务器把本次查询的结果缓存起来（无论查询缓存区的操作模式是什么，这个查询都不会被缓存）。

如果你想查询的数据表变动比较频繁，禁用查询缓存机制的做法可能更有实际意义。对于这类查询，查询缓存区说不定还会帮倒忙。比如说，很多Web站点都会把对其Web服务器的访问请求记录到一个数据表里，并定期对该数据表进行一些统计分析。如果这个Web站点办得还算不错，就会有源源不断的新数据行被插入到这个数据表里，所以查询缓存区中的查询结果也就会

相当快地变得与实际情况相脱节。这样，虽然你是在反复多次地发出同一个查询，但查询缓存区却并没有多大的实际价值。对于这类场合，用SQL\_NO\_CACHE修饰符向服务器表明你不想让它把查询结果缓存起来的做法当然更为明智。

#### 4.6.2 与硬件有关的优化问题

到目前为止，我们在这一章里讨论的都是些与硬件配置无关的优化措施。如果有更好的硬件，服务器当然能运行得更快。但并非所有的硬件升级都会有同样的优化效果。在对各种硬件升级方案进行评估的时候，最重要的原则仍是前面介绍如何调整服务器参数时所提到的：要把尽可能多的信息尽可能长时间地保存在快速存储设备上。

如果想通过改变硬件配置的办法来改善服务器的性能，可以从以下几个方面进行：

- **在机器里安装更多的内存。**内存多了，就能加大服务器的各种缓存区和缓冲区的容量，从而延长有关信息驻留在内存里的时间，减少磁盘读写活动。
  - **如果机器里的RAM足以支持数据交换工作全都发生在一个内存型文件系统里，就应该重新配置系统，去掉那些基于硬盘的数据交换设备。**否则，即便有充足的内存，有些系统也还是会使用磁盘来进行数据交换工作。
  - **购置高速磁盘以缩短I/O等待时间。**硬盘的磁头寻道时间对性能的影响最大。相对而言，磁头在磁道之间的移动是很慢的；在磁头移动到位之后，从磁道上把信息读出来则要快得多。不过，如果在扩充内存和购置高速硬盘之间做选择的话，还是应该选择扩充内存。内存永远比硬盘快；扩充了内存，就能加大缓冲区的容量，减少磁盘读写活动。
  - **把硬盘读写活动分散到多个物理设备上，提高并行操作能力。**与把信息都保存在同一个物理设备上相比，把它们分散到多个物理设备上的做法将加快其读写操作。比如说，如果把数据库安排在一个设备上，把日志信息安排在另一个设备上，就能并行地对这两个设备进行读写，这要比让数据库和日志共享同一个设备的做法快很多。但要提醒大家的是，同一物理设备上的不同分区并不能使性能得到多大的改善，因为它们仍要竞争同一个物理资源——磁头。对日志和数据库进行移动的具体做法请参见第10章中的有关内容。在对数据进行移动之前，必须先把系统上的负载情况弄清楚。如果某个物理设备上的负载已经比较大了，再把数据库搬到那儿说不定反而会使系统性能变得更糟。比如说，如果Web站点很繁忙，再把数据库搬到Web文档树所在的物理设备上就不会有任何性能改进。（顺便说一句，如果机器里只有一个硬盘驱动器，分散磁盘读写活动也就无从谈起了。）
- 使用RAID设备也可以带来一些并行操作方面的好处。
- **使用多处理器硬件。**对MySQL服务器之类的多线程应用程序而言，多处理器硬件可以同时执行多个线程。





## 第二部分 MySQL 程序设计接口

### 第5章 MySQL 程序设计简介

### 第6章 MySQL 应用程序设计接口：C语言

### 第7章 MySQL 应用程序设计接口：Perl DBI

### 第8章 MySQL 应用程序设计接口：PHP语言

## 第5章 MySQL程序设计简介

本章将讨论为什么要自行编写基于MySQL的程序而不是简单地使用MySQL发行版本自带的标准客户程序，并对后面几章里使用的三种程序设计语言（C、Perl、PHP）的MySQL编程接口以及选用这些语言时需要考虑的一些因素做了概括性的介绍。

### 5.1 为什么要自行编写MySQL程序

每一种MySQL发行版本都自带着一组工具程序，比如用来查看数据表结构和内容的mysqldump、用来把数据文件加载到数据表里去的mysqlimport、用来完成管理操作的mysqladmin以及用来与服务器进行交互并执行各种查询命令的mysql等等。这些MySQL标准工具程序的设计都非常精练，每个只负责实现一项特定的功能。即便是比其他工具程序更富变化的mysql——可以用它来执行各种不同类型的查询——也不例外：它只能用来直接向服务器发出SQL查询命令并查看其结果。

每个MySQL客户程序只负责一项具体工作的做法并不是一种设计缺陷，相反，这正是它的预定目标。期望这些通用性的工具程序能满足用户的一切需求是不现实的。MySQL的开发者们并不以“向用户提供一个包罗万象的程序”为目标，因为那只会毫无必要地增加代码量而使程序变得过于庞大和臃肿。

MySQL自带的客户程序能让MySQL用户完成各种最为常见的工作，但并不足以解决你在实际工作中可能会遇到的每一个问题。在本书的这一章里，将向大家介绍一些在自行编写能够访问MySQL数据库的程序时应该知道的东西。为帮助人们开发应用程序，MySQL准备了一个相当完备的客户程序开发库。不管你的应用程序在访问MySQL服务器方面有多么特殊的要求，都可以利用这个开发库把它实现出来。只要你想得到，MySQL客户程序开发库就做得得到。

与使用mysql客户程序去访问MySQL服务器的情况相比，自行开发的程序能够让你实现以下几个目标：

- **对输入进行定制。**如果使用的是mysql程序，就只能输入SQL语句。如果使用的是自行开发的程序，就能以更直观和更易于使用的方式进行输入。你用不着精通SQL，甚至连数据库的工作原理和具体用途都用不着关心。可以用最基本的命令行界面来显示提示信息和读取输入，可以用屏幕I/O函数库来实现数据表的全屏输入，还可以用curses或S-Lang函数库、使用Tcl/Tk语言的X窗口、Web页面等多种手段来获得更花哨的效果。

与不得不构思并写出一条正确的SELECT语句相比，只需填写检索条件的做法更受大多数人的欢迎。比如说，当一位房地产代理商想知道自己手里有没有在某个价格、房型或地点范围内的房子时，肯定希望只需输入几个数字就把事情搞清楚。数据的录入或修改工作也是如此；与不得不掌握INSERT、REPLACE、UPDATE等SQL语句的语法相比，数据录入人员肯定希望只需输入必要的数字。

在最终用户和MySQL服务器之间增加一个输入层的另一个好处是可以对用户提供的输入做一些预处理。比如说，日期值是不是MySQL所预期的格式、必须填写的项目是不是都填上了等等。

有些场合——比如MySQL的输入是来自其他程序时——根本不需要人去插手。比如说，Web服务器可以把运行日志直接写入MySQL数据库而不是文件，而一个定期运行的系统监控程序也可以把系统的状态信息记录到数据库里去。

- **对输出进行定制。**mysql程序的输出谈不上什么排版效果，它只提供了两种格式——以制表符加以分隔或表格格式。如果想让输出报告美观一点，就得亲自去排版。比如说，可以在输出报告里用“Missing”（意思是“缺失”）来表示NULL值，也可以生成复杂的报表。请看下面这份报表：

State	City	Sales
AZ	Mesa	\$94,384.24
	Phoenix	\$17,328.28
	subtotal	\$117,712.52
CA	Los Angeles	\$118,198.18
	Oakland	\$38,838.36
	Subtotal	\$157,036.54
	TOTAL	\$274,749.06

这份报表有以下几个地方值得注意：

- 定制的标题行。
- State列里的美国州名只出现一次。
- 增加了subtotal（小计）和total（总计）计算。
- 把94384.24之类的数值显示为美元金额\$94,384.24。

另一项需要进行复杂排版的常见工作是生成发票——需要把顾客资料、货物、金额等信息安排到适当的位置；而这类报表用mysql程序是很难排版的。

还有一些工作，比如需要用检索出来的信息进行计算并把计算结果插入到另一个数据表或者想把查询结果发送到其他地方而不是显示给进行查询的用户时，使用mysql程序反而多费周折。比如说，可以让程序在从数据库提取出人名和电子邮件地址之后自动生成群发邮件作为其输出（这些输出是给那些收件人而不是给运行程序的人看的），甚至可以让它把邮件都发送出去。

- **突破SQL自身的局限性。**SQL是一种非过程化程序设计语言，不具备条件语句、循环语句、子例程等流控制结构。构成SQL脚本的各个语句是按从头至尾、一次一个的方式顺序执行的，几乎没有出错检查和处理机制。

当使用mysql程序以批处理方式执行一个SQL脚本文件的时候，mysql要么会在第一次出错



时就立刻退出，要么（如果使用了--force选项的话）会一口气执行完全部语句而不管发生了多少次错误。如果是自行编写的程序，就可以增加一些流控制语句，让程序根据有关查询的执行情况而有选择地前进。比如说，可以根据前一个查询是否成功而决定是否要执行后一个查询，或者在完成前一个查询之后先对其结果做些处理再继续往下执行。

SQL语句的执行结果大都没有“持久性”，一开始执行后一条语句，前一条语句的执行结果就不复存在了；这一特点也被带到了mysql程序中：很难把前一个查询的结果用到后一个查询上，也很难把多个查询的结果集中到一起。

虽然可以用LAST\_INSERT\_ID()函数查知前一条语句所生成的AUTO\_INCREMENT值，也可以对SQL变量进行赋值并在稍后引用，但也仅此而已。

这种局限性使一些很常见的工作——比如先检索出一些记录再依次对它们做一些复杂的处理等——很难单独使用SQL来完成。比如说，当想先检索出一份顾客名单、再依次查出他们每一位的信用资料时，往往需要使用好几个查询才能处理完一位顾客。同样的道理，如果你的工作必须连续使用多个相互关联的查询才能完成，就不适合使用mysql程序来进行。总之，当必须使用几个彼此相关的查询才能解决问题或者需要对查询结果做比较复杂的排版时，自行开发一个mysql程序的替代品就很值得考虑；新程序的作用是把多个查询“串联”在一起，使你能够把前一个查询的输出用做后一个查询的输入。

- **把MySQL集成到应用程序里。**在我们周围，有很多程序因为利用了数据库系统的信息收集和检索能力而如虎添翼。比如说，当需要验证顾客的身份或者查看库存情况时，某个应用程序会发出一条数据库查询命令；当网上顾客要求查看某位作者的作品时，Web应用程序会去查询数据库并把检索结果发送至客户浏览器。

如果任务比较简单，利用shell脚本把MySQL“集成”到应用程序里的方案（编写一个shell脚本，让它先调用mysql程序去执行一个内容为SQL语句的输入文件，再用各种UNIX工具程序对它的输出做必要的处理）还是可以考虑的。但这种做法并不值得提倡，尤其是在任务变得更加复杂的时候。随着应用程序功能的增加，脚本变得越来越难维护，它在进程创建方面的开销也越来越大。直接与MySQL服务器进行交互要有效得多，可以在应用程序的各个执行阶段把准确的信息从数据库里提取出来，当应用程序功能扩充的时候，只需增加一个模块就行了。

在第1章里，我们列举了几个以样板数据库sampdb为例的编程目标，现在是开始实现这些目标的时候了。下面是我们打算自行编程实现的几项功能：

- 对“美国历史研究会”的会员名录进行排版和打印输出。
- 把会员名录放到网上供人们在线搜索。
- 通过电子邮件向会员发出续交会费通知。
- 简化考试记分项目中的成绩录入工作，让老师能够使用Web浏览器来进行录入。

现在，将向大家介绍一下如何把MySQL与Web环境集成在一起。MySQL不能直接支持Web应用，但如果把MySQL与一些软件工具结合起来，客户端的用户就能通过Web服务器发出数据库查询命令，而查询结果将被显示在该用户的浏览器里。这样，经由Web访问数据库就变得简单了。

MySQL与Web的结合可以用下面两句相互补充的话来描述:

- **Web服务器使对MySQL的访问更加方便。**这句话里的重点是数据库, Web被视为一种使数据库更容易访问的工具。这是MySQL管理员考虑问题时的立场。MySQL是这类场合里的主角, 程序开发工作将围绕数据库来做文章, 比如编写一些Web页面来查看数据库里都有哪些数据表、各个数据表都有怎样的结构、它们的内容又是什么等等。
- **MySQL使Web服务器的功能得到了加强。**这句话里的重点是Web站点, MySQL被视为一种使站点内容更有价值因而更吸引人们来访问的工具。这是Web站点开发人员考虑问题时的立场。Web站点是这类场合里的主角, 利用Web站点来做文章, 比如用数据库来保存人们发给公告板或讨论组的帖子等。MySQL在这类场合扮演的角色比较微妙, 到访Web站点的人们可能根本觉察不到MySQL的身影。

这两种说法并不对立。在“美国历史研究会”的例子中, 我们将把会员名录放到网上, 让人们能够通过Web来查阅它的内容; 这是把Web当做数据访问工具的说法。与此同时, 在“美国历史研究会”的网站上提供会员名录又增加了这个站点在人们心目中的价值; 这是把MySQL当做Web站点增值工具的说法。

不管你更喜欢哪一种说法, 把MySQL与Web集成在一起的具体实现工作并没有什么区别: 以Web服务器为纽带把充当前端的Web站点和充当后端的MySQL连接在一起。先由Web服务器收集来自网络客户的信息并以查询命令的形式把它发送给MySQL服务器, 再由MySQL服务器去进行检索, 最后由Web服务器把查询结果(可能还要经过一些处理)送往网络客户的浏览器去显示。

完全可以不把数据放到网上, 但把数据放到网上往往利大于弊。与通过标准的MySQL客户程序访问数据相比, 把数据放到网上有以下几个好处:

- 人们不必非得运行一个MySQL客户程序才能访问你的数据了, 他们现在可以使用任何一种Web浏览器在任何一种平台上做这件事。无论MySQL客户程序有多么流行, 也肯定比不上Web浏览器。
- Web界面往往比MySQL标准客户程序的命令行界面更易于使用。
- Web操作界面可以根据应用软件的具体情况进行定制, 而MySQL标准客户程序的操作界面很难改变。
- 动态Web页面扩展了MySQL数据库的能力, 能够完成一些单独使用MySQL客户程序很难或者无法完成的任务。比如说, 网上购物就是一个无法单独使用MySQL客户程序来实现的应用项目。

任何一种程序设计语言都可以用来编写基于Web的应用程序, 但有些语言比其他语言更适合这项工作, 将在第5.3节里讨论这个问题。

## 5.2 可用于MySQL的API

为帮助大家开发基于MySQL的应用程序, MySQL准备了一个用C语言编写的客户程序开发库, 这个开发库使人们能够在任何一个C语言程序里访问MySQL数据库。这个开发库实现了一个“应用程序设计接口”(application programming interface, API), 它对客户程序如何去建立



和实现与MySQL服务器的通信做出了规范。

但MySQL程序并非只能用C语言来编写。在编写能够与MySQL服务器进行通信的应用程序时，有好几种程序设计语言可供选择。有很多程序设计语言或者处理程序是用C语言写的，或者具备调用C语言函数库的能力。因为这些语言是建立在C语言基础上的，而MySQL客户程序开发库又是一种C API，所以它们都能利用这个开发库所提供的各种手段把MySQL与自己绑定在一起。Perl、PHP、Python、Ruby、C++、Tcl以及其他语言都有用来开发MySQL客户程序的API。供Java程序使用的编程接口也早已出现了（但它们直接实现了“客户/服务器”协议，不再使用C语言函数库来处理通信事宜）。适用于其他程序设计语言的新API在不断地涌现，可以在MySQL网站的开发者园地（网址如下）查到一份最新的名单：

<http://www.mysql.com/portal/development/html/>

每一种程序设计语言的MySQL API都定义有自己的编程接口，这些编程接口对如何使用这种语言去访问MySQL数据库做出了规定。限于篇幅，本书不可能对每一种MySQL API都做出详细的讨论，这里仅讨论以下三种最为流行的API：

- **配合C语言使用的API：MySQL客户程序开发库。**这是最基础的MySQL程序设计接口，MySQL发行版本中的标准客户程序（如mysql、mysqladmin、mysqldump等等）就是用它实现出来的。
- **配合Perl语言使用的API：DBI（Database Interface，数据库接口）。**DBI是Perl DBD（Database Driver，数据库驱动程序）模块中的一个子模块，它与其他子模块共同构成了Perl语言的DBD层，Perl脚本将通过它们去访问各种不同的数据库引擎。（有很多种DBD子模块，但这里我们最感兴趣的当然是负责提供MySQL访问支持的DBI模块。）DBI模块最常见的用途有两个：一是用来编写供人们在命令行上启动的独立型MySQL客户程序；二是用来编写供Web服务器以调用方式启动去访问MySQL数据库的模块型Perl脚本。
- **配合PHP语言使用的API：PHP API。**PHP是一种服务器端脚本开发语言，它使人们能够方便地在服务器端把程序嵌入到Web网页中。在被发往客户端之前，这种页面将先由服务器主机上的PHP解释器做相应的处理，这就使人们能够利用PHP脚本去生成一些动态的内容（比如把MySQL数据库的查询结果嵌入到页面里）。“PHP”的原意是“个人主页”（personal home page），但PHP的迅猛发展使它突破了这层含义。PHP官方网站现在把这个名字解释为GNU（GNU's Not UNIX，GNU不是UNIX）风格的回文句——“PHP: Hypertext Preprocessor”（PHP：超文本预处理器）。类似于DBI模块，除MySQL以外，PHP还提供有对其他几种数据库引擎的访问支持。

这三种API将分别在本书的第6、7、8章中详细介绍。本章主要对它们各自的基本特点进行对比，以便让大家能够根据应用程序的具体要求在它们当中做出选择。

不必死抱着一种API不放。对它们了解得越深，所做出的选择就会越明智。如果项目很大，组件很多，就是把这些API都用上也未尝不可：一些组件用这种语言来实现，另一些组件则用另一种语言来实现。为了加深对这些API的理解和丰富自己的程序设计经验，在时间允许的前提下，甚至可以采用多种不同的方案去实现同一个项目。

如果你还没有把使用这些API所必需的软件包安装到自己的机器里,请参阅本书附录A中的有关内容。

如果读者不满足于本书所介绍的MySQL程序设计知识,请自行参阅其他书籍。我最熟悉的两本书(因为我就是它们的作者!)是*MySQL and Perl for the Web* (New Riders, 2001)和*MySQL Cookbook* (O' Reilly, 2002)。前一本书对MySQL和DBI在Web环境里的应用做了深入的研究;后一本书讨论了Perl和PHP,并对如何使用Python语言的DB-API接口和Java语言的JDBC接口去编写MySQL程序做了介绍。

### DBI和PHP的前身

DBI的前身是Perl语言中的Mysqlperl模块。因为缺乏支持,希望大家在开发MySQL新项目时最好不要使用Mysqlperl模块。Mysqlperl是专为MySQL而开发的模块,DBI则不然。当你想把一个为MySQL编写的Perl应用软件移植到另一个数据库引擎上时,利用DBI编写出来的脚本——因为它们对具体的数据库引擎不那么依赖——要比利用Mysqlperl编写出来的脚本更容易移植。(那些通过Mysqlperl模块去访问MySQL的Perl脚本仍可用在DBI环境里,因为你在编译DBI模块的时候可以把对Mysqlperl模块的仿真支持功能收入其中。)

PHP 3和PHP 4的前身PHP/FI 2.0 (FI是“form formatter”的缩写,意思是“表单解释器”)。类似于Mysqlperl模块的情况,PHP/FI也已经过时,所以这里就不多讨论它了。PHP 3也正逐渐被PHP 4取代,后者有着更丰富的功能和更好的性能。

### 5.2.1 C API

C API用在C程序的编译执行环境里。这个函数库提供了与MySQL服务器进行对话所必需的最底层接口,使我们能够与MySQL服务器建立连接并进行通信。

MySQL发行版本里有很多用C语言写的客户程序,MySQL C API是这些程序的基础。不仅如此,这个API还是其他程序设计语言(但不包括Java)的MySQL API模块的基础。比如说,Perl DBI模块和PHP处理器里与MySQL有关的代码只有与MySQL C API链接之后才能真正具备访问MySQL数据库的能力。

### MySQL C API的起源

如果你有为mSQL RDBMS编写程序的经验,就会注意到MySQL C API与mSQL的C API非常相似。这并不是巧合。当初,在开始实现SQL引擎的时候,MySQL的开发者们注意到mSQL提供了一些既有用又免费的工具程序。为了降低把mSQL工具程序移植到MySQL上的难度,MySQL API被有意识地设计成与mSQL API差不多的样子。(MySQL甚至提供了一个名为mysql2mysql的脚本来完成mSQL API函数名到MySQL函数名的文字转换工作。这种转换虽然动作不大,效果却非常好,很多mSQL程序就是这样被移植到MySQL里来的。)

## 5.2.2 Perl DBI API

DBI API用在由Perl脚本构成的应用环境里。在我们将要学习的三种API里，DBI的体系结构是最先进的：在能够与很多种数据库引擎配合工作的同时，它把尽可能多的数据库操作细节隐藏了起来，极大地减轻了脚本编写者的负担。DBI是通过一组Perl模块的分工合作而做到这一点的，这些模块构成了一个分上、下两层的体系结构（如图5-1所示）：

- DBI（database interface，数据库接口）层负责向客户脚本提供通用的接口。这是一个不依赖于任何一种数据库引擎的抽象层次。
- DBD（database driver，数据库驱动程序）层通过各种驱动程序与具体的数据库引擎打交道，每种数据库引擎都有与之对应的驱动程序。为MySQL实现DBI支持的DBD层模块叫做DBD::mysql。这个模块以前叫做Mysql-Mysql-modules，因为它最初是为mSQL编写的，后来才被扩展到MySQL上。它现在的名字DBD::mysql反映出这样一个事实：MySQL现在比mSQL更为人所知。

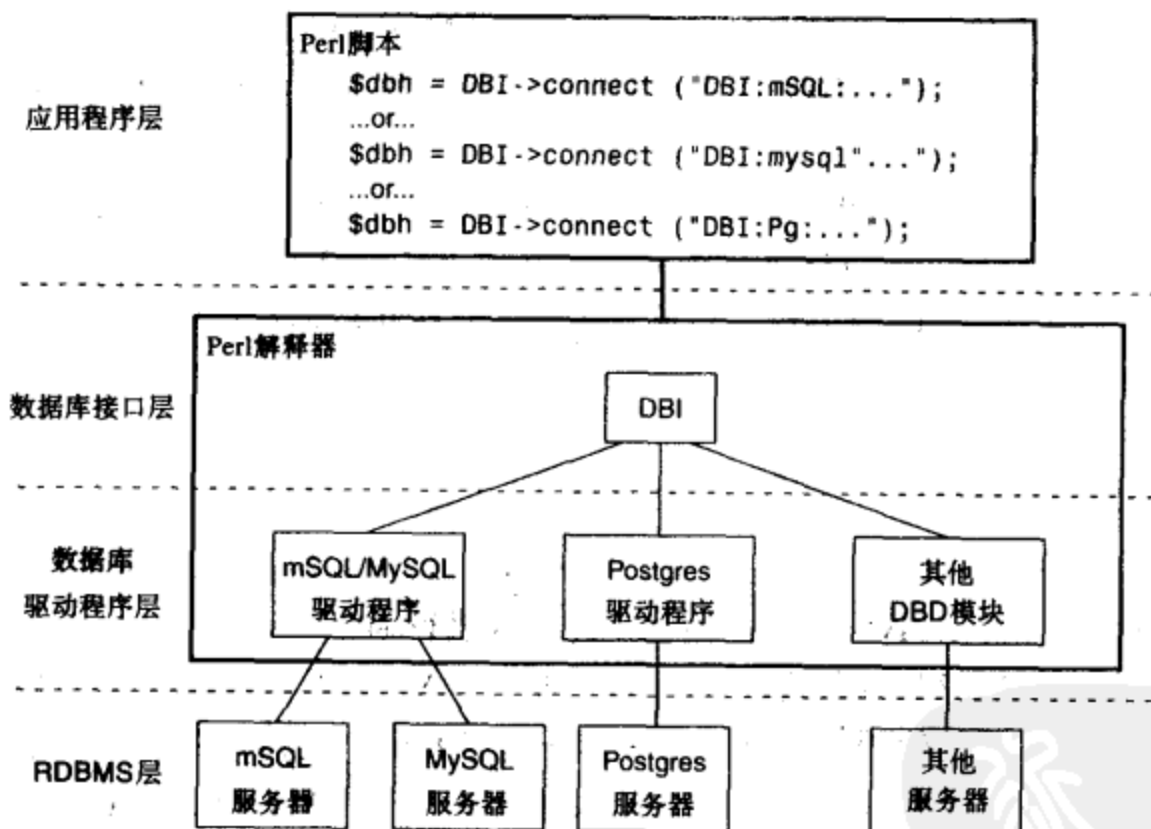


图5-1 DBI模块的体系结构

DBI模块的体系结构使我们能够以相当通用的格式来书写脚本代码。在编写DBI脚本时，使用一组格式固定的标准化函数去访问数据库。当脚本运行时，DBI层将在DBD层启动正确的驱动程序处理请求，而与数据库服务器打交道的具体细节都将由驱动程序负责。DBD层把从数据库服务器返回的数据传递给DBI层，再由DBI层传递给应用程序。数据的格式也是固定的，不管它们源自哪一种数据库。

从程序员的角度看，DBI API提供的编程接口既能掩盖各种数据库引擎之间的差异，又能与众多的数据库引擎（只要有相应的驱动程序就行）配合工作。这使程序员能够按一种固定的方式去访问不同的数据库，从而大大增加了DBI脚本的可移植性。

但你编写的DBI脚本必须知道你想让它连接到哪一个数据库去工作，因此，必须在用来连接数据库服务器的代码里明确地指定一个具体的数据库引擎。比如说，如果想使用一个MySQL数据库，就要像下面这样建立连接：

```
$dbh = DBI->connect ("DBI:mysql:...");
```

如果想使用PostgreSQL或mSQL数据库，就要像下面这样建立连接：

```
$dbh = DBI->connect ("DBI:Pg:...");
```

```
$dbh = DBI->connect ("DBI:mSQL:...");
```

把连接建立起来之后，就不必再提到具体的驱动程序或数据库引擎了。数据库方面的操作细节将全部由DBI和所指定的驱动程序负责处理。

上面讲的只是理论。在实践中，DBI脚本的可移植性还要受到下面两个因素的影响：

- 不同的RDMBS引擎对SQL语言的理解有着细微的差异，为这个引擎编写的SQL语句拿到另一个引擎上可能就无法执行。如果DBI脚本里的SQL语句都是普通的操作，把它移植到其他引擎应该没问题；但如果脚本里有依赖某具体引擎的SQL语句，就可能无法在其他引擎上使用了。比如说，如果在脚本里使用了MySQL独有的SHOW TABLES语句，就不能把这个脚本用在另一种数据库上了。
- 不同的数据库系统有着不同的特色功能。在DBI脚本里使用这些特色功能没有错，DBD模块因此而返回一些“特色”信息也没有错，但你必须知道DBI脚本的可移植性可能会因此而降低。比如说，MySQL DBD所返回的查询结果里还包含着数据列的几种属性，比如它的最大数据长度、它是不是数值型数据列等等；但其他数据库的DBD模块却不一定能提供这些属性。如果在DBI脚本里以MySQL DBD独有的办法使用了这些属性，就不能再把这个脚本移植到其他数据库上去了。

如果能够有意识地避开这两个问题，利用DBI API提供的数据库访问机制而编写出来的脚本就会有非常好的可移植性。别忘了，是否采用缺乏可移植性的方案是由你来决定的。在第7章中，读者将会发现其中并没有刻意避免使用MySQL DBD的独有功能（本书的附录G对它们都做了详细的说明）。这样做的目的是为了让大家对MySQL DBD的特色功能有所了解，以便在编写自己的DBI脚本时能够更好地做出是否需要使用它们的决定。

### DBI和DBD的含义

有些人把DBI看做是“database independent”（意思是“与数据库无关”）的缩写，把DBD看做是“database dependent”（意思是“与数据库有关”）的缩写。可以这样理解，但这并不是它们的本义。它们的真正含义是“database interface”（数据库接口）和“database driver”（数据库驱动程序）。

### 5.2.3 PHP API

类似于Perl，PHP也是一种脚本语言。但与Perl不同的是，PHP是一种专为编写Web应用程序而研制的编程语言，人们很少用它来编写通用性的脚本程序。PHP API的主要用途是把一些可



执行脚本嵌入到Web页面里去,使Web程序员能够很容易地编写出带有动态内容的页面来。当客户浏览器向Web服务器发出一个PHP页面请求时,Web服务器将先调用PHP去执行这个页面里的每一个脚本并把它替换为该脚本的输出,然后再把结果页面发送给浏览器。这就使该页面实际显示在浏览器里的内容能够根据当时的系统环境发生变化。比如说,如果把下面这个PHP小脚本嵌入到Web页面中,它就会把客户端主机的IP地址显示在浏览器里:

```
<?php echo $_SERVER["REMOTE_ADDR"]; ?>
```

PHP脚本的用途当然远不止此,完全可以利用PHP脚本把根据数据库里的当前内容而生成的最新信息提供给Web站点的到访者。下面是一个可以用在“美国历史研究会”网站上的PHP脚本示例。这个脚本将发出一个查询去统计最新的会员人数并把这个数字报告给访问该站点的人(如果执行出错,这段脚本代码将什么也不显示):

```
<html>
<head>
<title>U.S. Historical League</title>
</head>
<body bgcolor="white">
<p>Welcome to the U.S. Historical League Web Site.</p>
<?php
# USHL home page
$conn_id = @mysql_connect ("cobra.snake.net", "sampadm", "secret")
    or exit ();
mysql_select_db ("sampdb")
    or exit ();
$result_id = mysql_query ("SELECT COUNT(*) FROM member")
    or exit ();
if ($row = mysql_fetch_row ($result_id))
    print ("<p>The League currently has " . $row[0] . " members.</p>");
mysql_free_result ($result_id);
?>
</body>
</html>
```

PHP脚本看起来就像是一些把可执行代码嵌在“<?php”和“?>”标记之间的HTML页面。同一页面允许包含任意多个代码段,这就大大增加了脚本开发工作的灵活性。比如说,可以先把PHP脚本写成一个普通的HTML页面——即搭建起基本的页面框架,然后再逐步添加PHP代码去生成该页面中的动态内容部分。

PHP不像DBI那样使用一个统一的接口去访问不同的数据库引擎,它与数据库引擎之间的接口更像是一组用C语言实现的底层API函数调用,每种数据库引擎都对应着一组专用的接口函数。比如说,在用来访问MySQL数据库的PHP脚本里,负责完成数据库访问操作的PHP函数与MySQL C客户程序开发库里的等效函数有着相似的名字。(喜欢DBI风格的读者可以选用PEAR; PEAR是“PHP Extension and Add-on Repository”的字头缩写,意思是“PHP扩展和增值函数库”。PEAR对PHP进行了扩充,它包括一个名为PEAR::DB的模块,该模块通过一种与

DBI相类似的双层体系结构向程序员提供了一组更抽象的数据库引擎访问接口。详细情况请访问站点<http://pear.php.net>。)

### 5.3 选择API

本节将向大家提供一些关于如何根据应用项目的具体情况去选用API的指导意见。我们将从MySQL C API、DBI API和PHP API的能力对比入手，因为只有在了解了它们的长处和短处之后，才能做出最正确的选择。

在开始讨论之前，首先声明我对这几种API并无偏见——虽然我有自己的偏好。你们迟早也会有自己的偏好，就像本书的技术顾问们一样。事实上，本书的一位技术顾问认为我应该多强调一下C语言在MySQL程序设计工作中的重要性；另一位则认为我应该少写点C语言的事和别鼓励大家使用它！希望大家能够根据这一节的讨论内容得出自己的结论。

在为应用项目挑选API的时候，应该考虑以下几个因素：

- **预期的运行环境。**应用程序将被用在怎样的环境里。
- **性能。**用这种API语言编写的应用程序有着怎样的执行效率。
- **开发工作的难易程度。**用这种API及相关语言编写应用程序有多么困难或多么容易。
- **可移植性。**这个应用程序会不会被移植到MySQL以外的数据库系统上。

下面将对这几个因素分别加以探讨。需要注意的是，有些因素是相互制约的。比如说，你很想让应用程序执行得最有效，但能否迅速完成这个项目——哪怕在性能上稍微差一点——也同样重要。

#### 5.3.1 运行环境

在动手编写代码之前，应该对应用程序的运行环境有一定的了解。比如说，它可能是一个将由shell启动执行的报表生成程序，或者是一个在每个月的末尾作为一个cron作业运行的账户使用费统计程序，或者是一个将由Web服务器调用执行的模块。一般来说，由shell或cron启动执行的命令大都是一些独立的程序，很少用到来自运行环境的信息；由Web服务器调用执行的模块则大都需要从其运行环境了解一些特定的信息，比如：客户使用的是哪一种浏览器？客户在邮件列表订阅表单里都输入了哪些参数？客户访问其个人资料时输入的口令是否正确？

不同的API语言在不同的运行环境里有着不同的适用性，应该选用最适当的API来编写你的应用程序：

- C是一种通用语言。从理论上讲，可以用它来做任何事情。但在实际工作中，C语言多被用来编写可独立运行的程序而不是开发Web应用项目。导致这一现象的原因之一可能是C语言中的文本处理和内存管理工作不像Perl或PHP语言那样简单，而这两项工作在Web应用项目里却非常多。
- 类似于C语言，Perl也很适合用来编写可独立运行的程序，但它在Web站点的建设中也大有用武之地（比如说，通过CGI.pm模块）。这使Perl语言非常适合编写需要把MySQL和Web链接在一起的应用程序：与Web服务器的通信交给CGI.pm模块去负责，与MySQL的

通信交给DBI去负责。

- PHP是一种专为编写Web应用程序而研制的语言，因而也最适合用在这类环境里。同时，数据库访问也是PHP的一个强项。因此，当遇到需要访问MySQL数据库的Web应用项目时，选用PHP就是一件很自然的事情了。此外，还可以把PHP当做一个可独立运行的解释器来使用（比如说，从shell执行PHP脚本），但这种用法不多见。

根据以上分析，当需要编写可独立运行的应用程序时，C和Perl是最佳候选；当需要编写Web应用程序时，Perl和PHP是最佳候选。顺便说一句，如果你对这几种语言都不很精通可又必须编写两种类型的应用程序（不管它是独立型还是模块型），Perl应该是你的首选——因为它需要学习的东西最少。

### 5.3.2 性能

如果其他方面全都一样，程序当然是运行得越快越好。但性能的重要性与程序的使用频率也有着很大的关系。如果某个程序只在每个月末作为一个cron作业在晚上运行一次的话，就没必要苛求其性能。可如果某个程序必须在一个繁忙的Web站点上每秒运行好几次的话，最不起眼的改进也会使其性能得到大幅度的提升。性能的好坏往往决定着Web网站的生死。不管网站的内容有多好，只要它反应迟钝，就会影响到它的访问率。如果网站是你的一个收入来源，低劣的性能将直接导致利润的减少。如果你不能提供及时的服务，等得不耐烦的访问者就会离开你的站点到别处去。

性能评估是一项复杂的工作。如果你想知道用某种API写出来的应用程序执行得能有多快，最好的办法就是把它写出来并运行一下试试。如果你拿不准哪一种API最合适，最好的办法就是用这些不同的API把程序写出来并加以对比。当然，事情并非总得这样才能弄明白。比较常见的情况是这样的：首先把程序写出来，等它正常运转起来之后，再去考虑它的优化问题。比如说，它还能不能运行得再快点儿？内存还能不能再少用点儿？别的地方还能不能再改进点儿？但你必须知道有两个因素对性能的影响是很“稳定”的：

- 编译型程序要比解释型脚本执行得更快。
- 对于用在Web环境里的解释型语言来说，让Web服务器把这种语言的解释器当做它自己的一个模块去调用的做法要比把这个解释器运行作为一个独立进程的做法性能更好。

#### 1. 编译型语言与解释型语言

与用脚本语言编写的同功能程序相比，用编译型语言编写的程序在经过编译之后将更紧凑、内存耗用量更少、执行得也更快——因为它没有语言解释器在执行脚本程序时的开销。C语言是编译型的，Perl和PHP则是解释型的，所以C程序要比Perl或PHP脚本运行得更快。因此，那些使用频繁的程序应该用C语言来编写。

不过，有些因素能减少这种明显的差距。用C语言写出来的程序是执行得很快，但这并不代表不存在效率低下的C程序。用编译型语言来编写程序的做法并不是高效率的同义词，还需要把程序的具体用途也考虑进去。此外，如果一个以脚本程序为主的应用软件把它的大部分时间都用来执行MySQL客户程序开发库——这个库已经被链接到了一个解释器引擎里——的例程代码，编译型程序和解释型程序之间的性能差距也就不像想像中那么大了。

## 2. 独立型语言解释器与模块型语言解释器

在基于Web的应用程序里，脚本语言解释器通常以两种面目出现——至少对Apache（本书中的Web应用程序都是按照将与这种Web服务器配合使用而编写的）来说是这样的：

- 可以安排Apache把脚本解释器启动为一个独立的进程。在这种操作模式下，每当Apache需要运行一个Perl或PHP脚本时，它就会启动相应的解释器并让它去执行那个脚本。此时，Apache是把脚本解释器当做一个CGI程序来使用的——也就是说，Apache将使用CGI（Common Gateway Interface，公共网关接口）协议与它们进行通信。
- 可以把脚本解释器用做一个模块而直接链接到Apache的二进制代码里，使它运行成为Apache进程的一个组成部分。用Apache术语讲，Perl和PHP解释器将以mod\_perl和mod\_php模块的面目出现。

Perl和PHP的拥护者为这两种解释器谁快谁慢而争论不休，但都同意解释器以何种方式运行要比这两种语言本身的差异对性能的影响更大。当以模块形式运行时，这两种解释器都会比以一个独立CGI程序的形式运行时快得多。如果Apache把脚本解释器启动为一个独立的进程，那么，每执行一个脚本，就得启动一次解释器，这就大大增加了创建进程方面的开销。如果Apache把脚本解释器启动为一个模块，Web网页就能立刻得到解释器的处理。这就使系统的整体性能得到了显著的改善，开销的减少直接转换为性能的提高，加快了对外来请求的处理和响应速度。

独立型解释器的启动开销至少会使它的性能比模块型解释器差一个数量级。一般来说，Web页面的处理工作比较重视响应时间，计算量往往并不大。如果把这一因素考虑进去，因启动解释器而增加的开销就显得更大了。如果花了大量时间去启动解释器而只花了很少的时间去对脚本进行处理，无疑是在浪费系统的资源。这就好比花了一整天的时间去做准备，下午4点钟开始工作，而一到5点就收工回家了。

有些读者可能要问：解释器的模块型版本为什么会有这样的优点——毕竟，还必须启动Apache，对不对？这种节省是这样得来的：一个给定的Apache进程能够处理多个请求。当Apache启动时，它会立刻繁殖出一大堆子进程去处理外来的请求。当一个需要执行某个脚本的请求到达时，系统里早就有一个Apache进程在等着去处理它了。此外，Apache的每一个实例都能为多个请求提供服务，所以进程启动方面的开销是一组请求发生一次，而不是一个请求发生一次。

那么，当Perl和PHP都被安装为模块（即mod\_perl和mod\_php模块）形式时，它们中的哪一个执行得更快呢？这个问题争论已久，但随着PHP 4的出现，人们对这个问题的兴趣已经逐渐减弱了。PHP 3与Perl原本是有较大差距的：Perl会在运行脚本之前先把脚本代码转换为一种中间编译格式；PHP 3对脚本语句却是边解释边执行——这种做法太慢了，尤其是在脚本里有一个长循环的时候。PHP 4引入了一个名为Zend的高性能的解释器引擎，这个引擎使用了与Perl相类似的“先编译、后执行”模型。因此，只要有可能，就应该选用PHP 4而不是PHP 3。（这不仅是因为PHP 4改善了性能，而且还因为它实现了很多PHP 3不具备的语言功能。）

如果由你本人来安装PHP，强烈建议你选择PHP 4而不是PHP 3。如果你必须通过一个尚未进行过升级的账户或者服务商去使用PHP，可能只有PHP 3可用，但你应该要求服务商向你提供PHP 4。

Perl与PHP之间的另一个主要差异是Perl需要占用较多的内存；与mod\_perl模块链接的Apache进程要比与mod\_php模块链接的体积更大。人们在开发PHP时就已经考虑到它必须与其



他进程“生存”在一起，并可能会在该进程的生命期里被启动和禁用多次。Perl却是按照一个从命令行启动的独立型程序的思路来设计的，并没有被有意识地设计成一种将被嵌入到Web服务器进程里去的语言。这大概是它需要占用较多内存的一个原因——当被启动为一个模块时，Perl只是没有运行在适合它“生存”的环境里罢了。Perl的脚本缓冲机制和脚本用到的附加Perl模块是它需要占用较多内存的另一个原因，这两个原因都会导致有更多的代码进入内存并在Apache进程的生命期内一直驻留在那里。（人们已经开发出一些能够改善这一问题的技术。比如说，可以指定只有某几个特定的Apache进程允许激活mod-perl模块。这样，就只有这几个专门用来执行Perl脚本的进程会导致额外的内存开销。在Apache官方网站上的mod-perl园区里还有很多可供选择的策略，有兴趣的读者可访问<http://perl.apache.org/docs/>以获得更多的信息。）

语言解释器的独立型版本的确有一个模块型版本所不具备的优势：可以安排它把脚本运行在一个不同的用户ID下。模块型版本只能把脚本运行在与Web服务器是同一个账户的用户ID下，而出于安全方面的考虑，这个账户通常只具备最小的权限。这对那些需要有特定权限才能执行的脚本（比如那些需要对受保护文件进行读写的脚本）是很不利的。在条件允许的情况下，可以把模块型版本与独立型版本结合起来使用：默认使用模块型版本；如果脚本需要具备某个特定用户的权限才能运行，就切换使用独立型版本。

综上所述，不论选用的是Perl还是PHP，都应该尽量把语言解释器用做一个Apache模块而不是把它启动为一个独立的进程。但为了应付模块型版本无能为力的局面（比如脚本需要特殊权限才能执行），最好预备一个独立型解释器。这样，等真的遇到模块型版本无能为力的局面时，就可以利用Apache的suEXEC机制在给定用户ID下启动独立型解释器去处理那个脚本。（另一种最近才出现的解决方案是把Apache软件从1.x版本升级到2.x版本。Apache 2.x允许不同分组的脚本在不同的用户ID和用户分组ID下运行。）

### 5.3.3 开发周期

前面介绍了一些会影响应用程序性能的因素，但纯粹的执行时间可能并不是惟一的目标。时间以及编程工作的难易程度也是很重要的因素。因此，在挑选用来编写MySQL程序的API时，还应该考虑到怎样才能尽快地完成整个项目。如果只需花费一半的时间就能写出与C程序在功能上完全相同的Perl或PHP脚本，那么，哪怕编写出来的应用程序运行得不那么快，也很有可能不选择C API。较少关心程序的执行时间而更注重节约你本人的时间并非无理取闹，在将要编写的程序不会被频繁执行的场合就更是如此。要知道，你的时间要比机器的时间宝贵得多（人们发明机器的目的正是为了节约自己的时间）！

一般来讲，脚本语言可以更快地把程序编写出来，这就使它们特别适合用来搭建应用程序的框架。这至少是因为两个因素：首先，脚本语言提供的编程构造更先进，这使你能专注于问题的本质，把注意力集中在需要解决的问题而不是解决问题的具体细节上。比如说，当需要对呈“键字/键值”关系的数据（比如成对出现的“学生ID/学生姓名”）进行处理时，PHP语言中的关联数组和Perl语言中的散列将为你节约大量的时间。C语言就没有这样的构造。如果用C语言来实现这类构造，你不仅得亲自编写那些负责处理各种细节（比如内存管理和字符串操作）的代码，而且还必须亲自对它们进行调试。这一切都要花费你的时间。

其次，脚本程序的开发周期有较少的步骤。C程序的开发周期有“编辑-编译-测试”三个步骤；每修改一次程序，就必须重新对它进行编译和测试。Perl和PHP脚本的开发周期只有“编辑-测试”两个步骤，脚本在被修改后不需要进行编译就能立刻运行。但从另一方面讲，因为C编译器通过更苛刻的类型检查机制对程序的要求更为严格，所以它能帮你找出很多用松散型语言（比如Perl和PHP）无法发现的程序漏洞来。比如说，如果在C程序里拼错了一个变量名，C编译器就将向你提出警告，可PHP和Perl却不会给出任何提示，除非你要求它们这样做。随着应用程序在规模上的增长，对它进行维护将越来越困难。这时候，就能体会到那些苛刻要求的价值了。

总之，在编译型语言和解释型语言中做出选择其实就等于是在开发周期和程序性能上做出选择：用编译型语言开发出来的程序执行得较快，但需要花费较长的时间；用解释型语言开发出来的脚本执行得较慢，但需要花费的时间较少。你打算选哪一样呢？

把这两种方法结合起来的可能会更好。先编写一个脚本作为“草稿”，并利用这个快速搭建起来的应用程序框架来检验你的思路思路和算法是否正确。等事实证明这个程序的确有用且人们需要经常用到它的时候，再把它用一种编译型语言重写一遍。这样，相当于获得了两方面的好处——既能快速搭建起应用程序的框架，又能使最终产品有最好的性能。

严格地讲，Perl DBI和PHP API并不能赋予你超越C API的能力——如果你的Perl或PHP解释器里没有链接MySQL C客户程序开发库，就不能使用这两种API去访问MySQL数据库。虽然都能让你与MySQL服务器打交道，但集成了MySQL能力的C环境与集成了MySQL能力的Perl或PHP环境是有着很大差异的，应该根据具体的任务去选用对你最有帮助的API，比如下面这两种情况：

- **内存管理。**在C语言里，只要使用了需要为之动态分配内存的数据结构，就不可避免地要与malloc()和free()打交道。Perl和PHP能处理好这方面的事情。比如说，它们都允许数组尺寸自动加大，允许随意使用可变长度字符串而不必操心内存管理方面的细节。
- **文本操作。**在这三种语言中，Perl的文本处理能力是最强大的；PHP位居第二，但与Perl差距不大；C语言在这方面的表现相当初级，远远地落在第三。

虽说可以用C语言来编写自己的函数库并把有关的内存管理例程和文本处理例程等打包为便于调用的函数，但对它们进行调试以及确保有关算法正确高效却是你的责任。从就事论事的角度看，Perl和PHP中的有关算法更值得信赖，因为它们已经被很多人检视过了，在调试和效率方面都应该没问题。利用他人的成果可以大大节省你自己的时间。（从另一个角度看，如果Perl或PHP解释器里真的存在漏洞的话，你就只有默默忍受或者另谋出路。但在使用C语言编写程序的时候，对程序的行为却可以进行更细致的控制。）

这几种语言的“安全”程度也是互不相同的。C API提供了对数据库服务器的最底层访问接口，实施的安全策略也最少，所以它是最不“安全”的。如果在C程序里把API函数的调用顺序弄错了，运气好的人会看到一条“out of sync”（数据不同步）出错信息，而运气差的人就只能吞下程序崩溃的苦果了。Perl和PHP对你的保护要周到得多：没有按正确顺序去调用API函数的脚本将执行失败，但Perl或PHP解释器却不会崩溃。动态分配内存及其相关指针也是C程序发生崩溃的一个常见原因。Perl和PHP都能替你管理内存，所以你的脚本很少会有内存管理方面的漏洞。

开发周期与编程语言的外部支持量也有很大的关系：C语言的外部支持是以一些封装着MySQL C API例程的开发库的形式出现的，很容易使用。C和C++都有配套的开发库。Perl语言的外部支持——即各种Perl模块（这些模块在概念上与Apache模块最为接近）——在数量上无疑是最多的。人们甚至专门设立了一个名为CPAN（Comprehensive Perl Archive Network，智能化Perl档案网）的机构来帮助你迅速找到和获得各种Perl模块。利用各种Perl模块，甚至一行代码都不必写就能使用很多种函数。想让你的脚本把根据某数据库的内容而生成的报告作为电子邮件的附件发送给什么人吗？好办，只需从cpan.perl.org站点下载某个MIME模块，就能立刻具备生成电子邮件附件的能力。PHP的外部支持目前还达不到这种程度，但PHP 4正随着PEAR的进一步开发而不断完善着。

#### 5.3.4 可移植性

这里所说的可移植性指的是把一个为MySQL编写的程序移植到另一种数据库引擎上需要做多少修改以及修改工作的难易程度。你可能从未考虑过这个问题。然而，只有那些毫无远见的人才会信口说出“我决不会把这个程序用在MySQL以外的任何数据库上”的话。假如你换了份新工作而新公司的数据库系统与你以前使用的不一样，该怎么办？如果必须考虑可移植性因素，就应该知道这几种API在这方面的差异：

- DBI的可移植性最好，因为“与数据库无关”正是DBI API的设计目标之一。
- PHP的可移植性较差，因为它不像DBI那样为不同的数据库引擎都提供了同样的访问接口。针对不同数据库引擎的PHP函数其调用格式（函数名、参数个数、参数类型）大都与相应的底层C API函数差不多。虽然人们试图消除这种差异，但你至少仍得对与数据库有关的PHP函数名进行修改才行。可能还需要对有关函数在应用程序中的调用顺序做一些改变，因为针对不同数据库引擎的PHP接口在执行流程方面也不尽相同。如果你想让PHP脚本有最大的可移植性，最好的办法是使用前面提到的专为PHP语言开发的数据库抽象模块——PEAR。
- C API的可移植性最差。MySQL C API就是专为MySQL设计的。

当需要用同一应用程序去访问多个数据库系统时，“与数据库无关”意义上的可移植性就显得特别重要。这类工作可能很简单，比如只需要把数据从一个RDBMS移到另一个；也可能很复杂，比如需要组合来自多个数据库系统的信息而生成一个报表。

DBI和PHP都允许同时访问多个数据库引擎，也就是说，可以同时连接多个不同的数据库服务器并进行访问，而这些数据库服务器甚至可以在不同的主机上。但在对来自不同数据库系统的数据进行检索和处理的时候，DBI和PHP又有不同的特点，因而有着不同的适用性。DBI是这类场合里的首选，因为针对不同数据库系统的同功能DBI函数都有着同样的调用格式。比如说，假设需要在MySQL、mSQL和PostgreSQL数据库之间传输数据。如果使用的是DBI，就只有用来连接三个数据库服务器的三条DBI->connect()调用是必须有所区别的。如果使用的是PHP（但没有使用PEAR），脚本就会复杂得多，因为将不得不混杂着使用三组不同的读调用和三组不同的写调用。这时候，你肯定会想到用PEAR模块来消除数据库访问机制之间的差别，把三组读、写调用的格式统一起来。



## 第6章 MySQL应用程序设计接口：C语言

MySQL提供了一个用C语言写成的客户程序开发库，可以利用这个库来编写能够访问MySQL数据库的客户端程序。这个库定义了一个应用程序设计接口，其中包括以下几大类例程：

- 连接管理类例程，用来建立和断开与MySQL服务器的连接。
- 查询构造与执行类例程（用来构造数据库查询命令并把它们发送给MySQL服务器去执行）和结果集处理类例程（用来对从MySQL服务器返回的查询结果数据进行处理）。
- 状态和出错报告类函数，用来在API调用失败时了解出错的原因。
- 选项处理类例程，它们能帮你处理选项文件或命令行上的选项。

本章将详细讨论如何利用MySQL C客户程序开发库来编写自己的程序并使这些程序与MySQL发行版本自带的客户程序有着比较一致的调用界面和参数格式。假设大家对C语言程序设计有着一定的了解，但并不要求读者必须是一位专家。

这一章将大致地按照从非常简单到相当复杂的顺序给出一系列客户程序示例。前几个客户程序示例将开发并逐步完善一个基本的客户程序框架，这个框架除建立和断开与MySQL服务器的连接外什么事情也不做。（这样做的原因是：虽然MySQL客户程序的用途各不相同，但它们的共同特点却是它们都必须建立与MySQL服务器的连接。）这个客户程序框架的开发工作将按以下阶段进行：

- 仅包含连接建立代码和连接断开代码（client1）。
- 增加出错检查功能（client2）。
- 增加运行时获取主机名、用户名、口令等连接参数的功能（client3）。

最终得到的client3程序有着很好的通用性，可以把它用做开发其他客户程序的基础。在完成了它的开发工作之后，我们将先去学习如何处理各种查询。首先将探讨如何对硬编码SQL语句进行处理，然后再开发一段能够用来处理各种SQL语句的代码。在完成这部分的学习之后，我们将给client3程序增加一些查询处理代码，如此开发出来的程序（client4）与MySQL自带的mysql客户程序非常相似，可以用它来交互地发出数据库查询命令。

再往后，本章将向大家介绍如何使用MySQL 4版本新增的两项功能：

- 如何编写使用安全套接字层（Secure Socket Layer, SSL）协议通过安全连接与MySQL服务器进行通信的客户程序。
- 如何编写使用嵌入式MySQL服务器开发库libmysqld的应用程序。

最后，我们将分析并解决一些常见的问题，比如“怎样才能获得数据表结构的描述信息？”、“怎样才能把图像插入到数据库里？”等等。

本章只在必要时才对来自MySQL C客户程序开发库的函数和数据类型进行介绍，这些函数和类型的完整清单及使用方法可以在附录F里查到。附录F对MySQL C客户程序开发库的其他方面也做了一些介绍，可以在开发工作中随时参考。



本章中的示例程序都可以从网上获得，可以直接运行它们而用不着亲自敲入它们。它们都收录在sampdb发行版本里，可以在这个发行版本的capi目录里找到它们。具体的下载办法请参见附录A。

### 到哪里去找示例程序

在MySQL邮件列表上经常会看到这样的问题：“哪里能找到用C语言写的客户程序？”如果让我来回答这个问题的话，那当然是：“在这本书里！”不过，似乎有很多人都没注意到在MySQL自带的客户程序里就有几个是用C语言写出来的（例如mysql、mysqladmin和mysqldump）。MySQL有源代码形式的发行版本，可以从中发现很多值得借鉴的客户程序代码。如果你还没有这样做过，就请下载一份MySQL软件的源代码发行版本并到它的client目录里去看看。

## 6.1 客户程序的制作流程

利用MySQL C客户程序开发库编写出来的C语言程序要经过编译和链接等几个步骤才能执行，本节将对这些步骤进行介绍。不同的系统用来制作客户程序的命令往往会有一些差异，这一节里的命令可能需要稍做修改才能用在你的系统上。不过，这里介绍的基本原则却是相当通用的，它们应该能被套用在大部分客户程序上。

### 6.1.1 对系统的基本要求

既然是用C语言来编写MySQL客户程序，那么肯定需要一个C语言编译器。这一节里的示例将使用gcc编译器，它是UNIX系统上最常见的编译器。除程序源代码以外，还需要以下东西：

- MySQL头文件。
- MySQL C客户程序开发库。

MySQL头文件和MySQL C客户程序开发库是开发MySQL客户程序的基础。如果你的系统上没有安装它们，就应该设法获得它们。如果MySQL软件是从源代码或二进制发行版本安装的，客户程序开发支持组件就应该作为安装过程的一部分已经安装好了。如果是用RPM文件来安装MySQL软件的，就必须安装开发者RPM文件才能把这些支持组件安装到机器里。MySQL头文件和MySQL C客户程序开发库的获取办法可以在附录A里查到。

### 6.1.2 MySQL客户程序的编译和链接

在编译和链接客户程序的时候，可能需要把MySQL头文件和MySQL C客户程序开发库的存放位置明确地告诉给编译器，因为它们的安装位置通常不在编译器或链接器的默认搜索路径上。在下面的例子里，将假设MySQL头文件和MySQL C客户程序开发库分别安装在/usr/local/include/mysql和/usr/local/lib/mysql目录里。

在把源文件编译为目标文件的时候，要用-I选项把MySQL头文件的存放位置告诉编译器。

比如说，如果要把源文件myclient.c编译为目标文件myclient.o，应该使用一个如下所示的命令：

```
% gcc -c -I/usr/local/include/mysql myclient.c
```

在把目标文件链接为可执行代码的时候，要用-L选项和-lmysqlclient参数把客户程序开发库的存放位置和名称告诉链接器，如下所示：

```
% gcc -o myclient myclient.o -L/usr/local/lib/mysql -lmysqlclient
```

如果客户程序是由多个文件组成的，就要在链接命令行上把所有的目标文件都列举出来。

链接步骤经常会因“某某函数没有定义”之类的错误而失败，此时，需要增加一个-l选项把这个“某某函数”所在的函数库的名字告诉链接器。比如说，如果这类出错信息与compress()或uncompress()函数有关，就要用-lz或-lgz参数来告诉链接器到哪里去搜索zlib压缩工具库：

```
% gcc -o myclient myclient.o -L/usr/local/lib/mysql -lmysqlclient -lz
```

如果这类出错信息与floor()等数学函数有关，就要用-lm参数来链接C语言的数学函数库。可能还需要增加其他的函数库，比如说，在Solaris系统上可能还需要-lsocket和-lnsl参数。

从MySQL 3.23.21版本开始，可以利用mysql\_config工具程序来确定MySQL程序的编译和链接参数。比如说，这个工具程序可能会给出如下所示的输出：

```
% mysql_config --cflags
-I'/usr/local/mysql/include/mysql'
% mysql_config --libs
-L'/usr/local/mysql/lib/mysql' -lmysqlclient -lz -lcrypt -lnsl -lm
```

如果能把mysql\_config直接用在编译或链接命令里，就需要把它放到一对反引号里：

```
% gcc -c `mysql_config --cflags` myclient.c
% gcc -o myclient myclient.o `mysql_config --libs`
```

系统将先执行mysql\_config并用它的执行结果替换掉反引号中的命令文本，自动地为gcc提供必要的编译或链接参数。

程序几乎都要经过反复修改才能最终完成。如果在每次修改之后都要敲入这么长的命令去编译和链接它，就算不出现打字错误，你也会觉得麻烦吧？其实用不着这么辛苦，这些工作完全可以交给make命令去完成。如果还没使用make命令建立过程序，赶快来补一下课吧。假设正在开发一个名为myclient的客户程序，它由两个源文件main.c和aux.c以及一个头文件myclient.h组成。可以用一个简单的Makefile文件去建立这个程序，如下所示。注意：Makefile文件中的行缩进必须用制表符（键盘上的Tab键）来实现，如果使用的是空格，这个Makefile文件就不能工作了。

```
CC = gcc
INCLUDES = -I/usr/local/include/mysql
LIBS = -L/usr/local/lib/mysql -lmysqlclient
all: myclient
main.o: main.c myclient.h
    $(CC) -c $(INCLUDES) main.c
aux.o: aux.c myclient.h
    $(CC) -c $(INCLUDES) aux.c
```

```
myclient: main.o aux.o
    $(CC) -o myclient main.o aux.o $(LIBS)
clean:
    rm -f myclient main.o aux.o
```

有了Makefile文件，只需简单地敲入make就能开始这个程序的重建工作，make将显示并执行必要的编译和链接命令：

```
% make
gcc -c -I/usr/local/mysql/include/mysql myclient.c
gcc -o myclient myclient.o -L/usr/local/mysql/lib/mysql -lmysqlclient
```

与敲入长长的gcc命令相比，这种办法既简单又不容易出错。Makefile文件还使编译工作更容易控制和管理。比如说，如果需要在链接步骤增加几个函数库——比如数学函数库和压缩工具函数库，只需在Makefile文件中的LIBS行追加-lm和-lz即可：

```
LIBS = -L/usr/local/lib/mysql -lmysqlclient -lm -lz
```

如果还需要用到其他的函数库，那就把它们也追加到LIBS行上。此后，当运行make命令的时候，它将自动使用LIBS行的新设置值。

除直接编辑Makefile文件外，还有一种办法可以改变make变量，即在命令行上设定它们。比如说，假如C编译器名为cc而不是gcc，可以这样做：

```
% make CC=cc
```

如果有mysql\_config工具程序，还可以把它用在Makefile文件里，这样，就用不着把头文件和函数库的路径名硬编码在Makefile文件里了。可以使用如下所示的INCLUDES和LIBS行：

```
INCLUDES = ${shell mysql_config --cflags}
LIBS = ${shell mysql_config --libs}
```

当发出make命令的时候，它会先执行mysql\_config命令并且用它的执行结果作为相应的make变量值。\${shell}语法是GNU make支持的，如果你的make版本与GNU make无关，就可能需要使用另一种稍微不同的语法。

如果使用的是集成开发环境（integrated development environment, IDE），可能根本就不会看到或者用到Makefile文件，这取决于具体使用的是哪一种IDE。

## 6.2 客户程序1——连接到服务器

我们的第一个MySQL客户程序简单到了极点——它连接一个服务器、断开连接、退出。这一系列动作本身并没有多大的用处，但必须把它们弄明白，因为如果不能连接到一个服务器的话，就无法对任何一个MySQL数据库进行访问。连接MySQL服务器是每一个MySQL客户程序都必须做的事情，所以这里开发的连接建立代码肯定要出现在所编写的每一个客户程序里。此外，这个任务也使我们能以一个比较简单的问题作为出发点。后面将对这个客户程序做一些改进，使其能够完成一些更有用的事情。

第一个客户程序（client1）的代码全部放在一个名为client1.c的源文件里：

```

/* client1.c - connect to and disconnect from MySQL server */

#include <my_global.h>
#include <mysql.h>

static char *opt_host_name = NULL;      /* server host (default=localhost) */
static char *opt_user_name = NULL;      /* username (default=login name) */
static char *opt_password = NULL;      /* password (default=none) */
static unsigned int opt_port_num = 0;    /* port number (use built-in value) */
static char *opt_socket_name = NULL;    /* socket name (use built-in value) */
static char *opt_db_name = NULL;        /* database name (default=none) */
static unsigned int opt_flags = 0;      /* connection flags (none) */

static MYSQL *conn;                     /* pointer to connection handler */

int
main (int argc, char *argv[])
{
    /* initialize connection handler */
    conn = mysql_init (NULL);
    /* connect to server */
    mysql_real_connect (conn, opt_host_name, opt_user_name, opt_password,
                        opt_db_name, opt_port_num, opt_socket_name, opt_flags);
    /* disconnect from server */
    mysql_close (conn);
    exit (0);
}

```

这个源文件首先把头文件my\_global.h和mysql.h包括了进来。根据MySQL客户程序的用途,它可能还需要再把其他一些头文件包括进来,但这两个是最基本的:

- my\_global.h——负责把其他几个常用的头文件,比如stdio.h,包括到源文件里来。如果正在Windows系统上编译这个程序,它还会根据Windows兼容性的要求把头文件windows.h也包括进来。(你可能不会在Windows下编译程序,但如果打算对外发布代码的话,把这个头文件也包括进来将对那些需要在Windows下进行编译的人们有帮助。)
- mysql.h——定义了基本的MySQL常数和数据结构。

文件的包括顺序是非常重要的: my\_global.h应该在与MySQL有关的其他头文件之前最先被包括到源文件里来。

接着, client1程序声明了一组变量,这组变量依次对应着将被用来连接MySQL服务器的各种参数。在这个客户程序里,这些参数的值都硬编码在代码里并且全部都是默认值。稍后,我们将使用一种更灵活的办法来处理这些参数,即允许来自选项文件或命令行的参数值覆盖这些默认值。(这也正是变量名全都以“opt\_”开头的原因,这个前缀的意思是那些变量最终将通过命令参数来设定。) client1程序还声明了一个指向MYSQL结构的指针,这个MYSQL结构将充当连接句柄。



client1程序的main()函数负责建立和断开与服务器的连接。建立连接需要以下两个步骤:

1) 调用mysql\_init()函数获得一个连接句柄。当把NULL传递给mysql\_init()函数的时候,它将自动分配一个MYSQL结构,初始化之,然后返回一个指向它的指针。MYSQL数据类型是一个用来保存与连接有关的信息的结构。我们把这种类型的变量称为连接句柄。

2) 调用mysql\_real\_connect()函数建立与服务器的连接。mysql\_real\_connect()函数的输入参数有很多,它们的含义如下:

- 一个连接句柄指针——必须是mysql\_init()函数调用的返回值。
- 服务器主机——对这个值的解释与具体的操作平台有关。在UNIX系统上,如果给出的是一个字符串形式的主机名或者是一个数字形式的IP地址,客户程序将使用一个TCP/IP连接去连接该主机。如果给出的是NULL或"localhost",客户程序将使用一个UNIX套接字去连接运行在本地主机上的MySQL服务器。
- Windows系统对这个参数也会做出类似的解释,但会用TCP/IP连接来代替UNIX套接字。此外,在基于Windows NT的系统上,如果给出的主机名参数是"."或NULL,客户程序将首先尝试使用一个命名管道去连接本地服务器。
- 连接操作所使用的MySQL账户的用户名和口令——如果用户名是NULL,客户程序开发库将把登录名发送给MySQL服务器。如果口令是NULL,则不发送任何口令。
- 将在连接建立后被选取为默认数据库的数据库的名字——如果这个值是NULL,则不选取数据库。
- 端口号或套接字文件——端口号是供TCP/IP连接使用的。套接字名则是供UNIX套接字连接(如果使用的是UNIX系统的话)或命名管道连接(如果使用的是Windows系统的话)使用的。如果这个参数的值是0或NULL,客户程序将使用默认的端口号或套接字(或命名管道)名。
- 一个标志值——client1程序传递给这个参数的值是0,因为它没有用到任何特殊的连接选项。

可以在附录F里找到对mysql\_real\_connect()函数的详细介绍。比如说,附录F对主机名参数与端口号和套接字参数的交互以及允许用在连接标志参数里的各种选项做了进一步的说明。附录F还介绍了mysql\_options()函数的用法,在调用mysql\_real\_connect()函数之前,可以用mysql\_options()函数对其他一些与连接有关的选项进行设置。

调用mysql\_close()函数并把指向某个连接句柄的指针传递给它就可以断开连接。如果这个连接句柄是当初通过把NULL传递给mysql\_init()而自动分配的,mysql\_close()函数将在断开连接时自动释放这个句柄。

想试试client1程序?请按照本章前面所介绍的步骤对这个客户程序进行编译和链接,然后运行之。在UNIX系统上,要用以下命令来运行这个程序:

```
% ./client1
```

在UNIX系统上,如果shell的搜索路径里不包括当前目录("."),这条命令里的"./"就不能省略。如果这个目录在搜索路径里或者使用的是Windows系统,就可以省略命令名里的"./",如下所示:

```
% client1
```

client1 程序将连接服务器、断开连接、退出。虽说没做什么让人激动的事，可它毕竟是我们开发的第一个客户程序。不过，这仅仅是一个开始而已，因为它有两个重大的缺陷：

- 这个客户程序没有进行任何出错检查，所以我们无法知道它是不是真的能够工作！
- 连接参数（主机名、用户名等等）都是硬编码在源代码里的。要是它能让使用者通过选项文件或者命令行来设置这些连接参数就更好了。

这两个缺陷都不难弥补。我们将在接下来的几节里解决它们。

### 6.3 客户程序2——增加出错检查功能

第二个客户程序与第一个没有本质上的差异，但我们给它增加了一些出错检查功能，使它能够应付一些可能发生的错误。很多程序设计教科书都会“把出错检查部分留给读者们作为练习”，这大概是因为出错检查工作——让我们勇敢地面对它——确实很麻烦。

不管怎么说，让MySQL客户程序能够对出错情况进行检查并做出适当的处理是一种非常值得提倡的好习惯。为了让我们能够了解它们的执行情况并做出适当的处理，大多数MySQL C API函数都会返回一些状态信息，忽略这些信息只会给自己带来麻烦。如果没有在事先对出错处理做出安排，一旦所编写的程序出了问题，这边是怨声载道的用户，那边是自己也莫名其妙的错误，那时可就悔之晚矣。

以我们编写的第一个程序client1为例。怎样才能知道它真的连接上了MySQL服务器？可以检查服务器日志，看其中有没有与运行client1程序的时间相对应的Connect和Quit事件：

```
020816 21:52:14      20 Connect      sampadm@localhost on
                        20 Quit
```

如果看到的是一条“Access denied”（访问被拒绝）信息，就说明client1程序根本就没有建立起与服务器的连接：

```
020816 22:01:47      21 Connect      Access denied for user: 'sampadm@localhost'
                        (Using password: NO)
```

然而，client1程序本身不能告诉我们发生了哪一种情况。事实上，它没有这个本事——它没有做任何出错检查，所以连它自己也不知道发生了什么事情。这是不可接受的。绝不应该为了知道自己是否连接上了服务器而去查看服务器日志！我们现在就来弥补这个缺陷，给client1程序增加一些出错检查代码。

有返回值的MySQL C API函数用来表明自己是否执行成功的做法可以分为两种：

- 如果函数的返回值是一个指针，那么，非NULL指针表示成功，NULL表示失败。（在这里，NULL的含义是“一个C语言中的NULL指针”，而不是“一个MySQL数据库中的NULL数据列值。”）

以前面介绍过的mysql\_init()和mysql\_real\_connect()函数来说，如果它们返回的是一个连接句柄指针，则表明调用成功；如果它们返回的是一个NULL，则表明调用失败。

- 如果函数的返回值是一个整数，那么，0表示成功，非零值表示失败。注意：这个非零值

并不是某个特定的数值,比如-1。MySQL C API函数在调用失败时的返回值是不可预料的,不能保证是某个特定的数值。你可能见过别人用下面这样的代码来测试MySQL C API函数mysql\_XXX()的返回值,但这种做法是错误的:

```
if (mysql_XXX() == -1)          /* 这种测试是错误的 */
    fprintf (stderr, "something bad happened\n");
```

这种测试可能工作,也可能不能工作。MySQL C API从没说过函数调用失败时的非零返回值是某个特定的值,它只能保证那个返回值不是零。正确的返回值测试代码应该是下面这样的:

```
if (mysql_XXX() != 0)          /* 这种测试是正确的 */
    fprintf (stderr, "something bad happened\n");
```

下面这种测试也是正确的,它与上面的测试等价,但写起来稍微简单点:

```
if (mysql_XXX())               /* 这种测试是正确的 */
    fprintf (stderr, "something bad happened\n");
```

如果你看过一些MySQL软件本身的代码,就会发现它使用第二种测试的情况要更多一些。

并非每个API调用都会返回一个值。前面用过的mysql\_close()就是一个没有返回值的函数。(它会调用失败吗?调用失败了又怎样?在这条连接上的工作反正已经完成了。)

当某个MySQL C API函数调用失败的时候,可以用这个API里的mysql\_error()和mysql\_errno()函数来了解函数调用失败的原因。mysql\_error()函数的返回值是一个包含着出错信息的字符串,mysql\_errno()函数的返回值则是一个数值形式的出错代码。这两个函数的输入参数都是一个连接句柄指针。应该在错误发生后立刻调用它们,如果在它们的前面又发出了另一个会返回状态信息的API调用,从mysql\_error()或mysql\_errno()调用获取的出错信息就将是后一个API调用的。

一般说来,程序的使用者们大都认为出错信息字符串要比出错代码更好理解。因此,如果只想调用这两个函数中的某一个,建议使用mysql\_error()。为完整起见,本章中的示例将同时使用这两个函数。

现在,我们将根据以上讨论来编写第二个客户程序client2,它与client1程序很相似,但增加了适当的出错处理代码。源文件client2.c如下所示:

```
/*
 * client2.c - connect to and disconnect from MySQL server,
 * with error-checking
 */

#include <my_global.h>
#include <mysql.h>

static char *opt_host_name = NULL;    /* server host (default=localhost) */
static char *opt_user_name = NULL;    /* username (default=login name) */
static char *opt_password = NULL;    /* password (default=none) */
static unsigned int opt_port_num = 0; /* port number (use built-in value) */
```

```

static char *opt_socket_name = NULL;    /* socket name (use built-in value) */
static char *opt_db_name = NULL;        /* database name (default=none) */
static unsigned int opt_flags = 0;      /* connection flags (none) */

static MYSQL *conn;                    /* pointer to connection handler */

int
main (int argc, char *argv[])
{
    /* initialize connection handler */
    conn = mysql_init (NULL);
    if (conn == NULL)
    {
        fprintf (stderr, "mysql_init() failed (probably out of memory)\n");
        exit (1);
    }
    /* connect to server */
    if (mysql_real_connect (conn, opt_host_name, opt_user_name, opt_password,
                           opt_db_name, opt_port_num, opt_socket_name, opt_flags) == NULL)
    {
        fprintf (stderr, "mysql_real_connect() failed:\nError %u (%s)\n",
                 mysql_errno (conn), mysql_error (conn));
        mysql_close (conn);
        exit (1);
    }
    /* disconnect from server */
    mysql_close (conn);
    exit (0);
}

```

出错检查逻辑建立在mysql\_init()和mysql\_real\_connect()函数在调用失败时都将返回NULL的基础上。请注意:虽然client2程序会检查mysql\_init()的返回值,但在它真的调用失败时我们却没有调用任何出错报告函数。这是因为:一旦mysql\_init()调用失败,我们就不能想当然地认为连接句柄里包含着有意义的信息。mysql\_real\_connect()就不同了,在它调用失败的时候,连接句柄里虽然仍不会包含着对应于一个合法连接的信息,但却包含着一些可以传递给出错报告函数的诊断信息。仍可以把这个连接句柄传递给mysql\_close()去释放mysql\_init()已经为它自动分配的各种资源。(切记,千万不要把这个连接句柄传递给别的MySQL C API函数!因为它们大都只能在连接已经成功建立的前提下才能调用,你的程序可能会崩溃!)

编译并链接client2程序,然后运行它:

```
% ./client2
```

如果client2程序没有产生任何输出(就像上面这样),就表示连接已成功建立。如果看到的是下面这样的情况:

```
% ./client2
```

```
mysql_real_connect() failed:
```

```
Error 1045 (Access denied for user: 'sampadm@localhost' (Using password: NO))
```



则表示连接没有建立起来，但现在可以知道它为什么没有建立起来了。这个输出还意味着client1程序从来没有成功地建立起与服务器的连接。（因为client1使用的是同样的连接参数，既然client2程序的连接请求被服务器拒绝了，client1当然也是如此。）但我们在运行client1时并不了解这个情况，因为它根本就没有进行过出错检查。client2进行了检查，所以它能告诉我们出了问题。

知道出了问题总比不知道好，这也正是我们要对API函数的返回值进行检查的原因。这种故事在MySQL邮件列表上每天都会发生。典型的问题是：“为什么我一发出查询我的程序就崩溃了？”或者“为什么我的查询什么东西都没查出来？”造成这类问题的原因往往是所讨论的程序在发出查询命令之前没有检查连接是否已被成功地建立起来，或者在检索结果集之前没有检查服务器是否已经成功地执行了查询命令。因为程序没有进行出错检查，所以往往连程序员也不知道问题到底出在哪里。总之，千万不要想当然地认为每一个API调用都会成功。

本章中的其他程序示例都进行了出错检查，所编写的程序都应该这样做。这看起来好像是增加了编程工作量，但从长远来看，它反而是一种节约手段，因为不必无谓地花费大量的时间去追踪莫名其妙的问题。在本书的第7章和第8章里，将沿袭这里使用的出错检查方法。

现在，假设在运行client2程序的时候确实看到了一条“Access denied”（访问被拒绝）出错信息。怎样解决这一问题呢？一种办法是修改源代码，把连接参数的初始值全部改为能让你连接并访问MySQL服务器的值，然后重新编译之。这个方案多少有点道理——至少能连接上服务器了。但连接参数的值却仍是被硬编码在程序里的。我是反对这种做法的，特别是考虑到口令值的时候。（你也许会这样想：只要把程序编译成二进制的可执行代码，就能把口令隐藏起来。可是，别人只要用strings工具程序处理一下二进制代码，口令就无处遁形了。更不用说那些能够直接访问到源代码的人了，他们不费吹灰之力就能看到口令。）

在下一节里，我们将向大家介绍一种更灵活的服务器连接办法。但现在先来编写一个更容易使用的出错报告函数，因为今后会有很多地方要用到它。这里继续沿用同时报告MySQL出错代码和描述性出错信息字符串的做法，但不想像下面这样每次都得写出mysql\_error()和mysql\_errno()函数的调用代码：

```
if (...some MySQL function fails...)
{
    fprintf (stderr, "...some error message....\nError %u (%s)\n",
            mysql_errno (conn), mysql_error (conn));
}
```

用一个下面这样的实用函数来报告出错信息要简便得多：

```
if (...some MySQL function fails...)
{
    print_error (conn, "...some error message...");
}
```

print\_error()将打印一条出错信息并自动调用MySQL出错报告函数。有了print\_error()，就不用再写出一个长长的fprintf()调用语句了，这也将提高程序代码的可读性。另外，如果能把print\_error()编写得更细致，让它在conn输入参数是NULL的情况下也能做些有实际意义的事情，

我们甚至能够把它用在mysql\_init()调用失败的场合里。这样，我们的出错报告调用将有一个统一的格式，不会再像以前那样混杂——有些是fprintf()，另一些则是print\_error()。下面就是根据以上要求和讨论编写出来的print\_error()函数：

```
void
print_error (MYSQL *conn, char *message)
{
    fprintf (stderr, "%s\n", message);
    if (conn != NULL)
    {
        fprintf (stderr, "Error %u (%s)\n",
                 mysql_errno (conn), mysql_error (conn));
    }
}
```

有些读者可能会质疑这种做法：“并不是每一个需要进行出错检查的地方都必须调用出错报告函数。你故意夸大了出错报告工作的繁琐，而目的不过是想让你编写的实用函数显得更有价值而已。再说了，也用不着每次都去敲入出错信息打印语句——可以只写一次，然后用复制加粘贴的办法把它们拷贝到需要用到它们的地方。”这些说法的确有一定的道理，但我也有我的理由：

- 即便是采用复制加粘贴的办法，也是较短的代码更方便一些。
- 不管你是否喜欢在需要报告出错信息的时候每次都去调用那两个API函数，总是重复性地写出那么多的出错报告代码迟早会让你觉得厌烦。你会自觉或不自觉地走一些“捷径”，从而导致出错报告前后不一致。把出错报告代码打包在一个便于调用的实用函数里能帮你抵制这种诱惑，使代码前后一致，整齐划一。
- 如果后来又想修改出错信息的格式，只需修改一个地方肯定要比把程序从头到尾看一遍（甚至多遍）更省事。或者，如果后来又决定把出错信息写到一个日志文件里而不是把它们写到stderr设备上去（或者同时写到这两个地方），只需修改print\_error()函数也是更省事。这种安排能够减少很多不必要的错误（比如打字错误），而且能帮你抵制走“捷径”的诱惑，使代码前后一致，整齐划一。
- 在用调试器来调试自己编写的程序时，可以很方便地在出错报告函数里设置一个断点，使这个程序在每次检测到有错误发生的时候都能及时地切换到调试器里去。

基于上述考虑，本章后面内容里的程序示例都将使用print\_error()函数来报告与MySQL有关的问题。

## 6.4 客户程序3——运行时获取连接参数

在这一节里，我们将用一些更灵活的方案去代替把连接参数的默认值硬编码在程序里的做法，比如说，让用户能够在命令行上设定那些参数的值。前两个客户程序都有一个共同的缺陷，即连接参数是直接写在源代码里的。不管需要改变它们当中的哪一个或者哪几个，都不得不对源文件进行编辑并重新编译之。这可不太方便，要是程序还有其他使用者的话，事情将更麻烦。运行时设定连接参数的常要手段之一是使用命令行选项。比如说，MySQL软件自带的很多程序

都能接受长、短两种形式的命令行参数，如下表所示。

参 数	长选项格式	短选项格式
主机名	--host= <i>host_name</i>	-h <i>host_name</i>
用户名	--user= <i>user_name</i>	-u <i>user_name</i>
口令	--password 或 --password= <i>your_pass</i>	-p 或 -p <i>your_pass</i>
端口号	--port= <i>port_num</i>	-P <i>port_num</i>
套接字名	--socket= <i>socket_name</i>	-S <i>socket_name</i>

为了与MySQL软件自带的客户程序保持一致，下一个客户程序client3将接受那些同样的格式。这一点做起来并不困难，因为MySQL C客户程序开发库已经为我们准备了一些选项处理功能。此外，我们的客户程序还将具备从选项文件里提取信息的能力，这就使得可以把连接参数放到~/.my.cnf文件（即登录主目录里的.my.cnf文件）或者某个全局选项文件里去。这样，就用不着在调用这个程序的命令行上每次都设定那些选项了。MySQL C客户程序开发库能帮你完成寻找选项文件和从中提取各有关选项值的工作。只需在程序里增加几行代码，就可以让它变得支持选项文件机制——既然可以借助别人的智慧，就用不着去构思如何编写有关的代码了。（选项文件的语法在附录E里有详细的描述。）

在开始编写client3程序之前，我们先来看几个演示MySQL选项处理功能的小程序。这些程序能够让你体会到选项处理工作其实是多么简单，它并不会增加连接MySQL服务器和处理查询命令等操作的复杂性。

#### 6.4.1 访问选项文件的内容

从选项文件读出连接参数值的工作可以用load\_defaults()函数完成。load\_defaults()函数将去寻找选项文件、分析它们的内容以找出你感兴趣的选项组、改写程序的参数向量（即argv[]数组），它会来自那些选项组的信息以命令行选项的形式放在argv[]数组的开头部分，使这些选项看起来就好像是在命令行上给出的一样。这样，用不着修改原来使用的选项处理代码，程序就能在分析命令行选项的时候获得来自选项文件的连接参数。在argv[]里，来自选项文件的选项将出现在命令名的后面、其他参数的前面（注意，不是被追加在argv[]的末尾）。换句话说，在命令行上给出的连接参数将出现在argv[]的后半段，因而能够覆盖由load\_defaults()添加进来的选项（如果它们同名的话）。

下面这个小程序的名字是show\_argv，它演示了load\_defaults()函数的使用方法以及它是如何改变argv[]参数向量的：

```
/* show_argv.c - show effect of load_defaults() on argument vector */

#include <my_global.h>
#include <mysql.h>

static const char *client_groups[] = { "client", NULL };

int
```

```

main (int argc, char *argv[])
{
    int i;

    printf ("Original argument vector:\n");
    for (i = 0; i < argc; i++)
        printf ("arg %d: %s\n", i, argv[i]);

    my_init ();
    load_defaults ("my", client_groups, &argc, &argv);

    printf ("Modified argument vector:\n");
    for (i = 0; i < argc; i++)
        printf ("arg %d: %s\n", i, argv[i]);

    exit (0);
}

```

show\_argv程序里的选项文件处理代码由以下几个部分组成:

- client\_groups[]——这个字符串数组存放着一些选项文件组的名字, 而程序将去读取这些选项组里的选项。一般来说, MySQL客户程序至少应该把 "client" 放到这个数组里去 (它对应于[client]选项组), 可以在这个数组里列出任意多个选项组。这个数组的最后一个元素必须是NULL, 其作用是表明选项组名单到此为止。
- my\_init()——这个函数将为load\_defaults()调用做一些必要的初始化准备工作。
- load\_defaults()——负责读取选项文件。它有四个输入参数, 依次是: 选项文件名中的前缀 (这个前缀应该永远是"my")、存放着你感兴趣的选项组名单的字符串数组 (client\_groups)、程序的输入参数计数器 (&argc) 和输入参数向量 (&argv[]) 的地址。注意, 因为load\_defaults()会改变程序的输入参数计数器和输入参数向量的值, 所以不能直接传递它们的值, 必须传递它们的地址。特别是argv, 虽然它本身已经是一个指针了, 但也要把它传递为&argv, 即该指针的地址。

为了显示load\_defaults()函数对输入参数数组的作用效果, show\_argv程序将把这个数组打印两次: 第一次打印的是在命令行上给出的选项, 第二次打印的则是调用load\_defaults()函数之后的输入参数数组。

注意, 如果想看到load\_defaults()函数的工作情况, 就必须保证在登录主目录里存在着一个名为.my.cnf的文件, 而且这个文件的[client]选项组里必须有几个选项。(在Windows系统上, 可以使用C:\my.cnf文件。)假设这个文件有着如下所示的内容:

```

[client]
user=sampadm
password=secret
host=some_host

```

那么, 运行show\_argv程序将产生如下所示的输出:



```
% ./show_argv a b
Original argument vector:
arg 0: ./show_argv
arg 1: a
arg 2: b
Modified argument vector:
arg 0: ./show_argv
arg 1: --user=sampadm
arg 2: --password=secret
arg 3: --host=some_host
arg 4: a
arg 5: b
```

在show\_argv程序第二次打印出来的参数向量里，来自选项文件的设置值已经成为参数表的一部分了。你可能会在其中看到一些既不是在命令行上给出的也不是来自~/.my.cnf文件的选项。如果真的如此，应该能在某个系统级选项文件的[client]选项组里找到它们。之所以会出现这样的情况，是因为load\_defaults()函数实际要去读取几个选项文件。在UNIX系统上，在读取登录主目录里的.my.cnf文件之前，它会先到/etc/my.cnf文件和MySQL数据目录中的my.cnf文件里去寻找选项。在Windows系统上，load\_defaults()函数将依次读取Windows系统目录里的my.ini文件、C:\my.cnf文件和MySQL数据目录里的my.cnf文件。

需要使用load\_defaults()函数的客户程序几乎都会把"client"放到选项组名单里去（这是为了获得各选项文件对MySQL客户程序的基本设置），但完全可以让选项文件处理代码从其他选项组里获取选项。比如说，如果能让show\_argv程序去读取[client]和[show\_argv]选项组里的选项，只需先把源文件show\_argv.c里的下面这行代码：

```
const char *client_groups[] = { "client", NULL };
```

改写为如下所示的样子：

```
const char *client_groups[] = { "show_argv", "client", NULL };
```

再重新编译show\_argv程序就能达到目的，新show\_argv程序将去读取[client]和[show\_argv]选项组里的选项。我们来验证一下，先在~/.my.cnf文件里增加一个[show\_argv]选项组：

```
[client]
user=sampadm
password=secret
host=some_host

[show_argv]
host=other_host
```

然后运行show\_argv程序，应该看到一个如下所示的输出，它与我们前面看到的不一样：

```
% ./show_argv a b
Original argument vector:
arg 0: ./show_argv
arg 1: a
arg 2: b
Modified argument vector:
```

```

arg 0: ./show_argv
arg 1: --user=sampadm
arg 2: --password=secret
arg 3: --host=some_host
arg 4: --host=other_host
arg 5: a
arg 6: b

```

选项值在参数数组里的先后次序是由它们在选项文件里的先后次序决定的，选项组的名字在client\_groups[]数组里的先后次序对此没有任何影响。因此，在选项文件里，应该把供客户程序专用的选项组放在[client]组的后面。这种安排的好处是：如果在两个选项组里都设定了某个选项，供程序专用的设置值就将覆盖掉[client]组里的设置值。可以在刚才的例子中看到这一点：[client]和[show\_argv]选项组都对host选项进行了设置，但因为[show\_argv]选项组位于选项文件的最后，所以它给出的host选项值在show\_argv程序的参数向量里的出现位置更靠后并因此而成为有效的选项值。

load\_defaults()函数不读取通过环境变量给出的设置值。如果需要用到MYSQL\_TCP\_PORT或MYSQL\_UNIX\_PORT等环境变量的值，就必须通过getenv()函数自行做出安排。这一章里的客户程序都没有用到环境变量。下面这段代码演示了如何对两个标准的MySQL环境变量值进行检查：

```

extern char *getenv();
char *p;
int port_num = 0;
char *socket_name = NULL;

if ((p = getenv ("MYSQL_TCP_PORT")) != NULL)
    port_num = atoi (p);
if ((p = getenv ("MYSQL_UNIX_PORT")) != NULL)
    socket_name = p;

```

在标准的MySQL客户程序里，环境变量值的优先级要低于通过选项文件或命令行给出的选项设置值。如果想在自己的程序里检查环境变量并与这一做法保持一致，就应该把环境变量的检查工作安排在调用load\_defaults()函数或者处理命令行选项之前（而不是之后）。

### load\_defaults()与系统安全

在多用户系统上，有些实用工具（比如ps程序）能够把任何进程（由其他用户启动执行的也包括在内）输入参数表显示出来。那么，这类具备一定的进程嗅探功能的实用工具会不会导致用load\_defaults()函数从选项文件读出并放到程序的输入参数表里去的口令泄密呢？不会，请不要为此而困扰，因为ps程序只显示argv[]数组最初的内容。由load\_defaults()创建的口令参数将指向内存中一个专门为它分配的区域，这个区域并不是argv[]向量的初始组成部分，所以ps程序是看不到口令的。

不过，在命令行上给出的口令却会被ps程序显示出来，所以建议大家不要在命令行上给出口令。为进一步降低安全方面的风险，可以让程序在开始执行之初就把口令从输入参数表里去掉，其具体做法将在下一节里介绍。

### 6.4.2 处理命令行参数

有了load\_defaults()函数,就能把所有的连接参数都放到客户程序的输入参数向量里,接下来的事情就是对这个向量进行处理了。handle\_options()函数就是为这一目的而设计的,它是MySQL C客户程序开发库中的一个API函数,只要链接了这个函数库,就可以使用handle\_options()函数了。

这一节介绍的选项处理方法最早出现于MySQL 4.0.2版本。在这之前,MySQL C客户程序开发库用来进行选项处理的代码是以getopt\_long()函数为基础的。如果用来编写MySQL程序的开发库来自某个早于4.0.2的MySQL版本,就需要使用getopt\_long()函数来处理命令行参数,有关这方面的详细讨论请参阅本书第1版中的这一章内容。可以在那本书的配套Web站点(<http://www.kitebird.com/mysql-book/>)上找到这一章,它是以PDF格式存放的。

现在,基于getopt\_long()函数的选项处理代码已逐步被取代为一个基于handle\_options()函数的接口。新的选项处理例程在以下几个方面做了改进:

- 能够更精确地设定合法选项值的类型和取值范围。比如说,不仅可以指定某个选项必须是整数值,还可以指定它必须是正整数且必须是1024的倍数。
- 增加了帮助文本,使调用一个标准库函数来打印帮助信息的工作更简便易行。不必再专门编写一些代码来产生帮助信息了。
- 内建有对--no-defaults、--print-defaults、--defaults-file、--defaults-extra-file等选项的支持机制。附录E对这些选项做了比较详细的介绍。
- 支持一组标准的选项前缀(如--disable-和--enable-),使布尔型(开/关)选项更便于使用。本章内容没有用到这些选项,请参考附录E中的关于选项处理的讨论。

**注意** 虽然新的选项处理例程最早出现于MySQL 4.0.2版本,但最好不要在4.0.5之前的版本里使用它们。因为在4.0.2到4.0.5版本之间的过渡期里,人们发现它们存在一些漏洞并进行了修补。

为了让大家对MySQL的选项处理机制有一个完整的认识,将在这一小节编写一个show\_opt程序。这个程序将调用load\_defaults()函数来读取选项文件、设置输入参数向量,然后用handle\_options()函数对它们进行处理。

可以利用show\_opt程序来试验连接参数的各种设定方法(比如通过选项文件、通过命令行等等),它会把最终用来连接MySQL服务器的参数值显示出来。show\_opt程序能帮助大家掌握下一个客户程序client3的工作情况,因为client3其实就是把这里的选项处理代码与前面开发的服务器连接代码结合起来而已。

show\_opt程序将执行以下操作来告诉我们在参数处理的各个阶段发生了哪些事情:

- 1) 把主机名、用户名、口令以及其他连接参数初始化为它们的默认值。
- 2) 把初始连接参数和输入参数向量的值打印出来。
- 3) 调用load\_defaults()函数读取选项文件的内容并改写输入参数向量,然后把修改后的结果向量打印出来。

4) 调用选项处理例程handle\_options()去处理输入参数向量, 然后把连接参数值的最终结果以及最后剩在输入参数向量里的东西打印出来。

下面是show\_opt程序的源文件show\_opt.c的内容, 我们随后将对它的工作流程进行说明。

```

/*
 * show_opt.c - demonstrate option processing with load_defaults()
 * and handle_options()
 */

#include <my_global.h>
#include <mysql.h>
#include <my_getopt.h>

static char *opt_host_name = NULL;      /* server host (default=localhost) */
static char *opt_user_name = NULL;      /* username (default=login name) */
static char *opt_password = NULL;      /* password (default=none) */
static unsigned int opt_port_num = 0;    /* port number (use built-in value) */
static char *opt_socket_name = NULL;    /* socket name (use built-in value) */

static const char *client_groups[] = { "client", NULL };

static struct my_option my_opts[] =      /* option information structures */
{
    {"help", '?', "Display this help and exit",
     NULL, NULL, NULL,
     GET_NO_ARG, NO_ARG, 0, 0, 0, 0, 0, 0},
    {"host", 'h', "Host to connect to",
     (gptr *) &opt_host_name, NULL, NULL,
     GET_STR_ALLOC, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
    {"password", 'p', "Password",
     (gptr *) &opt_password, NULL, NULL,
     GET_STR_ALLOC, OPT_ARG, 0, 0, 0, 0, 0, 0},
    {"port", 'P', "Port number",
     (gptr *) &opt_port_num, NULL, NULL,
     GET_UINT, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
    {"socket", 'S', "Socket path",
     (gptr *) &opt_socket_name, NULL, NULL,
     GET_STR_ALLOC, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
    {"user", 'u', "User name",
     (gptr *) &opt_user_name, NULL, NULL,
     GET_STR_ALLOC, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
    { NULL, 0, NULL, NULL, NULL, NULL, GET_NO_ARG, NO_ARG, 0, 0, 0, 0, 0, 0 }
};

my_bool
get_one_option (int optid, const struct my_option *opt, char *argument)
{

```



```
switch (optid)
{
case '?':
    my_print_help (my_opts);    /* print help message */
    exit (0);
}
return (0);
}

int
main (int argc, char *argv[])
{
int i;
int opt_err;

printf ("Original connection parameters:\n");
printf ("host name: %s\n", opt_host_name ? opt_host_name : "(null)");
printf ("user name: %s\n", opt_user_name ? opt_user_name : "(null)");
printf ("password: %s\n", opt_password ? opt_password : "(null)");
printf ("port number: %u\n", opt_port_num);
printf ("socket name: %s\n", opt_socket_name ? opt_socket_name : "(null)");

printf ("Original argument vector:\n");
for (i = 0; i < argc; i++)
    printf ("arg %d: %s\n", i, argv[i]);

my_init ();
load_defaults ("my", client_groups, &argc, &argv);

printf ("Modified argument vector after load_defaults():\n");
for (i = 0; i < argc; i++)
    printf ("arg %d: %s\n", i, argv[i]);

if ((opt_err = handle_options (&argc, &argv, my_opts, get_one_option)))
    exit (opt_err);

printf ("Connection parameters after handle_options():\n");
printf ("host name: %s\n", opt_host_name ? opt_host_name : "(null)");
printf ("user name: %s\n", opt_user_name ? opt_user_name : "(null)");
printf ("password: %s\n", opt_password ? opt_password : "(null)");
printf ("port number: %u\n", opt_port_num);
printf ("socket name: %s\n", opt_socket_name ? opt_socket_name : "(null)");

printf ("Argument vector after handle_options():\n");
for (i = 0; i < argc; i++)
    printf ("arg %d: %s\n", i, argv[i]);

exit (0);
}
```

show\_opt程序所实现的选项处理机制涉及以下几个方面,使用MySQL C客户程序开发库去处理命令行选项的程序几乎都要按照这个套路来进行:

1) 除my\_global.h和mysql.h头文件外, my\_getopt.h文件也要包括进来, 这个文件对MySQL选项处理例程的调用接口进行了定义。

2) 定义一个元素为my\_option结构的数组, 即show\_opt.c文件里的my\_opts数组。这个数组中的每一个my\_option结构分别对应着该程序能够识别的一个选项, 其内容是与该选项有关的各种信息, 比如选项的短名字和长名字、默认值、是一个数值还是一个字符串等等。my\_option结构中的各个成员稍后介绍。

3) 在调用load\_defaults()函数读完选项文件并改写好输入参数向量之后, 调用handle\_options()函数去处理那些选项。handle\_options()函数的前两个参数分别是这个程序的输入参数计数值和输入参数向量。(注意: 所传递的必须是这些变量的地址而不能是它们的值, 这与load\_options()函数的情况很相似。)第3个参数指向刚才定义的那个元素为my\_option结构的数组。第4个参数是一个指针, 它指向一个辅助函数。借助于handle\_options()函数和my\_option结构, MySQL C客户程序开发库能自动完成绝大多数的选项处理动作, 但有些特殊动作处理不了, 所以你的程序应该再定义一个辅助函数(即show\_opt.c文件里的get\_one\_option()函数)供handle\_options()调用。get\_one\_option()函数的操作情况稍后介绍。

MySQL C客户程序开发库使用my\_option结构来保存关于选项本身的信息, 每一个能被MySQL程序识别出来的选项都对应着一个my\_option结构, 下面是my\_option结构的定义。

```
struct my_option
{
    const char *name;           /* 选项的长名字 */
    int id;                     /* 选项的短名字或代号 */
    const char *comment;        /* 对选项的描述, 将显示在帮助信息里 */
    gp_ptr *value;              /* 选项的值将存放在这个变量里 */
    gp_ptr *u_max_value;        /* 该选项由用户定义的最大值 */
    const char **str_values;     /* 该选项合法可取值的数组(目前未使用) */
    enum get_opt_var_type var_type; /* 选项值的类型 */
    enum get_opt_arg_type arg_type; /* 选项值是否必需 */
    longlong def_value;         /* 该选项的默认值 */
    longlong min_value;         /* 该选项的最小值 */
    longlong max_value;         /* 该选项的最大值 */
    longlong sub_size;          /* 选项值的偏移量 */
    long block_size;            /* 选项值的倍数调整幅度 */
    int app_type;               /* 保留, 供特定应用程序使用 */
};
```

下面对my\_option结构的各个成员做一下介绍:

- name

长选项名。它是以--name形式给出的选项, 但不包括前导的连字符。比如说, 长选项--user在my\_option结构里将被写为“user”。

- id

短（单字符）选项名或一个与该选项相关联的代号（如果选项没有单字符名字的话）。比如说，短选项-u在my\_option结构里将被写为'u'。如果选项只有长名字而没有单字符名字，可以指定一个代号作为其仅供内部使用的短名字。代号必须是独一无二的，而且不能与现有的单字符名字相同。（为了满足后一条要求，不妨把代号都设置得比255（即单字符值的最大可取值）还要大。在本章后面的第6.7节里有一个采用这一技巧的例子。）

- comment

一个描述该选项用途的字符串。这是出现在帮助信息里的文本。

- value

这是一个gptr（通用指针）类型的值。它指向一个变量，API调用将把选项的参数放到这个变量里。在处理完选项之后，只需检查这个value成员，就能知道它的设置值。如果选项不带参数，value成员将是NULL。如果选项带参数，value成员所指向的那个变量的数据类型必须与var\_type成员的值保持一致。

- u\_max\_value

这也是一个gptr类型的值，但只能由服务器程序使用。对于客户程序，请把u\_max\_value设置为NULL。

- str\_values

这个成员目前尚未投入使用。在未来的MySQL版本里，可以把选项的合法取值放到这个成员里，即要求用户给出的选项值必须与合法取值之一相匹配。

- var\_type

选项值（即在命令行上紧跟在选项名后面的那个值）的类型。它有以下可取值：

var_type的可取值	含 义
GET_NO_ARG	没有选项值
GET_BOOL	布尔值
GET_INT	整数值
GET_UINT	无符号整数值
GET_LONG	长整数值
GET_ULONG	无符号长整数值
GET_LL	长长整数值
GET_ULL	无符号长长整数值
GET_STR	字符串值
GET_STR_ALLOC	字符串值

GET\_STR和GET\_STR\_ALLOC的区别是：如果var\_type取值为GET\_STR，选项变量将直接指向输入参数向量里的那个值；如果var\_type取值为GET\_STR\_ALLOC，系统将给该参数制作一个副本，而选项变量将指向这个副本。

- arg\_type

选项名后面是否需要跟着选项值。它有以下可取值：

arg_type的可取值	含 义
NO_ARG	选项名后面不跟着选项值
OPT_ARG	选项名后面的选项值允许省略
REQUIRED_ARG	选项名后面必须跟着选项值

如果arg\_type成员的取值是NO\_ARG，var\_type成员就必须被设置为GET\_NO\_ARG。

- def\_value

该选项的默认值。对于数值型的选项，如果没有在输入参数向量里明确地给这个选项设定一个值，则将使用这个值作为默认值。

- min\_value

该选项的最小值。对于数值型的选项，这将是它能够指定的最小值。比这个值还小的值将被自动取为这个值。0表示“没有最小值”。

- max\_value

该选项的最大值。对于数值型的选项，这将是它能够指定的最大值。比这个值还大的值将被自动取为这个值。0表示“没有最大值”。

- sub\_size

选项值的偏移量。用来调整数值型选项的取值范围：来自输入参数向量的选项值减去sub\_size之后的结果才是系统内部使用的该选项的值。比如说，如果命令行上给出的选项值取值范围是1~256，但程序内部实际使用的选项值取值范围却是0~255，sub\_size成员就将被设置为1。

- block\_size

选项值的倍数调整幅度。对于数值型的选项，如果这个值不等于0，则代表着一个乘法基数；输入参数向量里的选项值（如有必要）将向下舍入为与自己最为接近的这个基数的一个整数倍数。比如说，如果选项值必须是一个偶数，就要把block\_size成员设置为2，handle\_options()调用将把奇数的选项值向下舍入为与它最为接近的偶数。

- app\_type

保留，供特定应用程序使用。

每一个合法的选项都在my\_opt数组里有一个对应的my\_option结构。作为这个数组的结束标记，这个数组里的最后一个元素将是如下所示的样子：

```
{ NULL, 0, NULL, NULL, NULL, NULL, GET_NO_ARG, NO_ARG, 0, 0, 0, 0, 0, 0 }
```

当调用handle\_options()函数去处理输入参数向量的时候，它将跳过向量中的第一个参数（即程序名），从第二个参数开始对真正的选项参数（即那些以“--”开头的参数）进行处理。处理工作将一直进行到这个向量的末尾或者遇到一个特殊的“终结者”参数为止，“终结者”参数是连续的两个短划线字符（即“--”本身）。在遍历输入参数向量的过程中，每遇到一个选项，handle\_options()就会调用事先安排的那个辅助函数（show\_opt程序里的这个辅助函数叫做get\_one\_option()）去进行相应的处理。handle\_options()将向这个辅助函数传递三个参数：短选项名、一个指向对应于这个选项的my\_option结构的指针、一个指向这个选项的选项值（即输入



参数向量里紧跟在这个选项后面的那个值)的指针——如果这个选项不带选项值,第三个参数就将是一个NULL指针。

当handle\_options()返回的时候,输入参数向量里将只剩下那些不是命令选项的参数,输入参数计数值也将相应地减少为一个反映现在这种情况的数值。

下面是show\_opt程序某次运行的输出结果(假设~/.my.cnf文件的内容仍是第6.4.1节里最后一次运行show\_argv程序时的样子):

```
% ./show_opt -h yet_another_host --user=bill x
Original connection parameters:
host name: (null)
user name: (null)
password: (null)
port number: 0
socket name: (null)
Original argument vector:
arg 0: ./show_opt
arg 1: -h
arg 3: yet_another_host
arg 3: --user=bill
arg 4: x
Modified argument vector after load_defaults():
arg 0: ./show_opt
arg 1: --user=sampadm
arg 2: --password=secret
arg 3: --host=some_host
arg 4: -h
arg 5: yet_another_host
arg 6: --user=bill
arg 7: x
Connection parameters after handle_options():
host name: yet_another_host
user name: bill
password: secret
port number: 0
socket name: (null)
Argument vector after handle_options():
arg 0: x
```

在上面的输出里,来自命令行的主机名将取代选项文件里设定的值,用户名和口令则仍使用来自选项文件的默认值。虽然我们混合使用了短选项格式(-h yet\_another\_host)和长选项格式(--user=bill),handle\_options()还是正确地完成了对这些选项的分析工作。

辅助函数get\_one\_option()是为了配合handle\_options()函数而编写的。它在show\_opt程序里并没有多大的作用,只对--help或-?选项(此时,handle\_options()传递给get\_one\_option()的optid参数的值是'?)做了一下处理。下面是辅助函数get\_one\_option()的代码:

```

my_bool
get_one_option (int optid, const struct my_option *opt, char *argument)
{
    switch (optid)
    {
        case '?':
            my_print_help (my_opts);    /* print help message */
            exit (0);
        }
    return (0);
}

```

my\_print\_help()是一个来自MySQL C客户程序开发库的API函数，其输入参数是一个选项名，它将根据这个选项名以及相应的my\_option结构里的注释字符串（即comment成员）生成一条帮助信息。如果想看看它的工作情况，可以试试下面这条命令，它生成的帮助信息将出现在show\_opt程序输出结果的末尾部分：

```
% ./show_opt --help
```

根据具体情况，还可以给get\_one\_option()函数增加一些case语句。比如说，可以利用这个函数来处理口令选项。如果把口令选项的my\_option结构中的arg\_type成员设置为OPT\_ARG，那么，当使用口令选项的时候，既可以给出口令值，也可以不给出口令值。也就是说，可以把口令选项写成--password或--password=your\_pass的长格式形式，或者把它写成-p或-p your\_pass的短格式形式。MySQL客户程序通常允许在命令行上省略口令值，等运行时再提示你输入之。不在命令行上写出口令值的好处是不会让别人看到口令。在本章后面的程序示例里，我们将用get\_one\_option()函数来检查是否在命令行上给出了口令值。如果给出了口令值，就把它保存起来；如果没有给出口令值，就设置一个标志，让客户程序在开始连接MySQL服务器之前提示你输入一个口令。

利用现在这个机会，还可以对show\_opt程序中的几个my\_option结构中的有关成员进行一些修改，看看它们对show\_opt程序的行为到底有什么样的影响。比如说，如果把--port选项的my\_option结构中的min\_value、max\_value、block\_size成员分别设置为100、1000、25，然后重新编译并运行这个程序，将发现无法把端口号设置为从100~1000这个范围以外的某个值，并且所给出的端口号值将自动舍入为与之最为接近的25的整数倍数。

show\_opt程序实现的选项处理机制还能自动完成对--no-defaults、--print-defaults、--defaults-file、--defaults-extra-file等选项的处理。请大家自己去试试这些选项，看看会发生什么事情。

### 6.4.3 把选项处理机制融合到MySQL客户程序里

在这一节里，将从show\_opt程序里把那些纯粹是为演示选项处理API函数的用法而存在的“累赘”去掉，把剩下的精华部分融合到一个具有实用价值的MySQL客户程序里，使那个客户程序能够使用你通过选项文件或者命令行给出的各种选项去连接一个MySQL服务器。客户程序client3就是我们的开发成果，它的源文件client3.c如下所示：

```

/*
 * client3.c - connect to MySQL server, using connection parameters
 * specified in an option file or on the command line
 */

#include <string.h>      /* for strdup() */
#include <my_global.h>
#include <mysql.h>
#include <my_getopt.h>

static char *opt_host_name = NULL;      /* server host (default=localhost) */
static char *opt_user_name = NULL;      /* username (default=login name) */
static char *opt_password = NULL;      /* password (default=none) */
static unsigned int opt_port_num = 0;   /* port number (use built-in value) */
static char *opt_socket_name = NULL;    /* socket name (use built-in value) */
static char *opt_db_name = NULL;        /* database name (default=none) */
static unsigned int opt_flags = 0;      /* connection flags (none) */

static int ask_password = 0;            /* whether to solicit password */

static MYSQL *conn;                     /* pointer to connection handler */

static const char *client_groups[] = { "client", NULL };

static struct my_option my_opts[] =     /* option information structures */
{
    {"help", '?', "Display this help and exit",
     NULL, NULL, NULL,
     GET_NO_ARG, NO_ARG, 0, 0, 0, 0, 0, 0},
    {"host", 'h', "Host to connect to",
     (gptr *) &opt_host_name, NULL, NULL,
     GET_STR_ALLOC, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
    {"password", 'p', "Password",
     (gptr *) &opt_password, NULL, NULL,
     GET_STR_ALLOC, OPT_ARG, 0, 0, 0, 0, 0, 0},
    {"port", 'P', "Port number",
     (gptr *) &opt_port_num, NULL, NULL,
     GET_UINT, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
    {"socket", 'S', "Socket path",
     (gptr *) &opt_socket_name, NULL, NULL,
     GET_STR_ALLOC, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
    {"user", 'u', "User name",
     (gptr *) &opt_user_name, NULL, NULL,
     GET_STR_ALLOC, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
    { NULL, 0, NULL, NULL, NULL, NULL, GET_NO_ARG, NO_ARG, 0, 0, 0, 0, 0, 0 }
};

```

```

void
print_error (MYSQL *conn, char *message)
{
    fprintf (stderr, "%s\n", message);
    if (conn != NULL)
    {
        fprintf (stderr, "Error %u (%s)\n",
                 mysql_errno (conn), mysql_error (conn));
    }
}

my_bool
get_one_option (int optid, const struct my_option *opt, char *argument)
{
    switch (optid)
    {
    case '?':
        my_print_help (my_opts);      /* print help message */
        exit (0);
    case 'p':                          /* password */
        if (!argument)                /* no value given, so solicit it later */
            ask_password = 1;
        else                          /* copy password, wipe out original */
        {
            opt_password = strdup (argument);
            if (opt_password == NULL)
            {
                print_error (NULL, "could not allocate password buffer");
                exit (1);
            }
            while (*argument)
                *argument++ = 'x';
        }
        break;
    }
    return (0);
}

int
main (int argc, char *argv[])
{
    int opt_err;

    my_init ();
    load_defaults ("my", client_groups, &argc, &argv);

    if ((opt_err = handle_options (&argc, &argv, my_opts, get_one_option)))

```



```

    exit (opt_err);

    /* solicit password if necessary */
    if (ask_password)
        opt_password = get_tty_password (NULL);

    /* get database name if present on command line */
    if (argc > 0)
    {
        opt_db_name = argv[0];
        --argc; ++argv;
    }

    /* initialize connection handler */
    conn = mysql_init (NULL);
    if (conn == NULL)
    {
        print_error (NULL, "mysql_init() failed (probably out of memory)");
        exit (1);
    }

    /* connect to server */
    if (mysql_real_connect (conn, opt_host_name, opt_user_name, opt_password,
        opt_db_name, opt_port_num, opt_socket_name, opt_flags) == NULL)
    {
        print_error (conn, "mysql_real_connect() failed");
        mysql_close (conn);
        exit (1);
    }

    /* ... issue queries and process results here ... */

    /* disconnect from server */
    mysql_close (conn);
    exit (0);
}

```

与此前开发的client1、client2和show\_opt程序相比，client3程序多了几个新本事：

- 可以在命令行上选定一个数据库作为连接建立起来之后的默认数据库，只要在其他参数的后面多给出一个数据库名就行了。这种行为与MySQL软件自带的标准客户程序的做法是一致的。
- 如果输入参数向量里包含着一个口令值，get\_one\_option()函数将在对它进行复制后立刻擦掉那个原本。这将把别人有机会使用ps等系统状态查看程序而看到命令行上给出的口令的时间窗口压缩到最小的程度。（注意：这种做法只能把那个时间窗口压缩到最小的程度，但并不能彻底消除之。在命令行上给出口令的做法仍是一种安全风险。）
- 如果在命令行上只给出了口令选项而没有给出口令值，get\_one\_option()函数将设置一个标

志让程序在运行时提示你输入一个口令，这项工作将由main()函数在所有选项都被处理完毕之后调用get\_tty\_password()函数去完成。get\_tty\_password()是一个来自MySQL C客户程序开发库的API函数，它在提示你输入口令的时候不会把口令回显在屏幕上。有些读者可能会问：“为什么不用getpass()函数来做这件事？”因为getpass()函数并非在所有的系统上都能使用，比如说，它在Windows系统上就不能用；get\_tty\_password()则不同，这个API函数会根据所使用的系统平台做出调整，因而有着良好的跨系统移植性。

client3程序将使用所给出的各有关选项去连接MySQL服务器。在没有选项文件来“添乱”的前提下，如果在运行client3程序的时候没有给出任何选项，它将连接到本地主机localhost并将把UNIX登录名和一个空口令发往服务器；如果用下面这条命令去运行client3程序的话，它将提示你输入一个口令（因为-p选项的后面没有紧跟着的口令值）、连接到some\_host主机、把用户名some\_user和键入的口令发往服务器：

```
% ./client3 -h some_host -p -u some_user some_db
```

client3程序还会把数据库名some\_db传递给mysql\_real\_connect()，使它在连接建立起来之后被选取为当前的默认数据库。在有选项文件来“添乱”的情况下，client3程序将对选项文件的内容进行处理并根据处理结果对连接参数做出相应的修改。

在经过这么多的努力才得到的client3程序里，有些东西是每一个MySQL客户程序都不可缺少的，即用适当的参数去连接MySQL服务器的机制。这一机制是由客户程序框架client3.c实现的，可以把它当做编写其他程序的基础。拷贝它并添加一些具体的细节，就能得到一个有实用价值的客户程序。这使你能够把精力集中到真正感兴趣的东西（即访问数据库的内容）方面来。在编写的应用程序里，真正有用的操作都将发生在mysql\_real\_connect()和mysql\_close()调用之间。以这个基本框架为基础，我们就能迅速开发出不同用途的MySQL客户程序。现在，当需要编写一个新的客户程序时，可按照以下步骤去做：

- 1) 先拷贝一份client3.c文件。
  - 2) 如果会用到一些client3.c未曾涉及的其他选项，请对选项处理循环进行必要的修改。
  - 3) 在mysql\_real\_connect()和mysql\_close()调用之间插入特定的应用代码。
- 至此，一切万事大吉了。

## 6.5 查询的处理

我们连接MySQL服务器的目的是为了能够与它进行通信并交流信息。本节的学习重点是怎样把SQL查询命令发送给MySQL服务器去执行并对它返回的查询结果进行处理。每个查询都要通过以下几个步骤才能得到妥善的处理：

- 1) 构造查询命令。完成这一步骤的方式取决于查询命令本身的内容——具体地说，就是要看它本身是否包含二进制数据。
- 2) 把查询命令发送给服务器去执行。服务器将执行这个查询并生成一个结果集。
- 3) 对结果集进行处理。具体做法要由所发出的查询的类型来决定。比如说，SELECT语句通常都会返回一些数据行让你去处理，INSERT语句却不会这样。

在构造查询命令的时候，将用哪个函数把它们发往服务器也是考虑因素之一。最常用的查询发送函数是mysql\_real\_query()。这个例程把查询命令当做一个计数字符串（一个字符串加上一个长度值）来对待，必须计算出查询命令字符串的长度并把这个长度值随查询命令字符串一起传递给mysql\_real\_query()。因为查询将被视为一个计数字符串而不是一个以NULL字节结尾的字符串，所以它可以容纳包括二进制数据和NULL字节本身在内的任何东西。

查询命令也可以用mysql\_query()函数发出，这个函数用起来要简便一些，但它对查询命令字符串的限制要多一些。传递给mysql\_query()函数的查询命令必须是一个以NULL字节结尾的字符串，这就意味着查询命令本身不得包含NULL字节——查询命令字符串里的NULL字节将导致它被错误地解释，因为这种NULL字节在效果上等于是缩短了真正的查询命令。一般说来，如果查询命令可以包含任意二进制数据，就有可能包含NULL字节，也就不应该使用mysql\_query()。但从另一方面讲，如果查询命令肯定是一个以NULL结尾的字符串，就可以利用C语言标准函数库中的字符串函数（比如strcpy()和sprintf()等）来构造它们，这些函数应该是你已运用得非常熟练的了。

在构造查询命令字符串的时候，还必须考虑到特殊字符的转义问题。如果所构造的查询命令字符串包含有二进制数据或者要用到引号、反斜线等具有特殊含义和作用的字符，就必须对它们进行转义处理，将在第6.9.2节里对这一问题做进一步的讨论。

下面是查询处理机制的一个简单框架：

```
if (mysql_query (conn, query) != 0)
{
    /* failure; report error */
}
else
{
    /* success; find out what effect the query had */
}
```

如果查询命令执行成功，mysql\_query()和mysql\_real\_query()都将返回零；否则，它们都将返回一个非零值。所谓“成功的”查询，指的是MySQL服务器认为没有语法错误因而能够执行的查询命令，与查询命令的执行效果没有任何关系。比如说，成功的SELECT查询不见得会返回一些数据行，而成功的DELETE语句也不见得真的删除了某些数据行。查询命令的实际执行效果要用其他手段来检测。

查询会因为各种原因而失败。下面是一些比较常见的失败原因：

- 本身有语法错误。
- 在语义上有错误——比如，在查询命令里用到了一个其实并不存在的数据列。
- 查询命令将要访问某个数据表，但你却没有足够的权限去访问那个数据表。

可以把查询命令粗略地划分为两大类：一类是不会返回结果集的，另一类则是会返回结果集的。INSERT、DELETE、UPDATE等语句进行的查询都属于“不会返回结果集”那一类——即使对数据库做出了修改，它们也不会返回任何数据行，惟一能够得到的信息是有多少个数据行受到了它们的影响。

SELECT和SHOW等语句进行的查询都属于“会返回结果集”那一类，而人们在发出这类查询时也的确抱着它们将返回一些东西的希望。在MySQL C API里，这类查询所返回的结果集将被表示为MYSQL\_RES数据类型。这种数据类型其实是一个结构，其中容纳着各有关数据行的数据值以及关于这些数据值的元数据（metadata），如数据列的名字和数据值的长度等。结果集允许为空，即允许结果集里的数据行个数是零。

### 6.5.1 处理无结果集的查询

要想对一个无结果集的查询进行处理，首先要通过mysql\_query()或mysql\_real\_query()调用把它发送给服务器去执行。如果查询成功，就可以调用mysql\_affected\_rows()函数去查知它实际插入、删除、修改了多少个数据行。

下面是一个用来对无结果集的查询进行处理的代码框架：

```
if (mysql_query (conn, "INSERT INTO my_tbl SET name = 'My Name'") != 0)
{
    print_error (conn, "INSERT statement failed");
}
else
{
    printf ("INSERT statement succeeded: %lu rows affected\n",
           (unsigned long) mysql_affected_rows (conn));
}
```

请注意：这段代码是把mysql\_affected\_rows()函数的返回值转换为unsigned long值之后才把它打印出来的。mysql\_affected\_rows()函数的返回值是一个my\_ulonglong类型的值，但这种类型的值在某些系统上无法直接打印出来。（比如说，我曾在FreeBSD系统上成功地直接打印过这种值，但在Solaris上就不行。）作为一种通用的解决方案，需要把这个返回值转换为unsigned long类型并使用“%lu”作为打印格式符。这一解决方案也适用于其他一些会返回my\_ulonglong值的函数，比如mysql\_num\_rows()和mysql\_insert\_id()等。如果想让编写的客户程序具备跨系统的可移植性，千万要记住这个技巧。

mysql\_affected\_rows()的返回值能告诉你有多少数据行受到了它的影响，而这个“影响”的具体含义还要取决于所发出的查询命令的类型。对INSERT、REPLACE、DELETE语句来说，这个数字是指它们插入、替换、删除了多少个数据行。对UPDATE语句来说，这个数字指的是它实际修改了多少个数据行，即有多少个数据行真的因为这条语句而发生了变化——如果数据行在修改前后内容没有发生变化，MySQL将认为它没有被修改。也就是说，即使某个数据行符合UPDATE语句的WHERE子句所给出的选取条件，也并非一定会发生改变。

在某些场合，我们需要知道UPDATE语句到底“匹配到了多少个数据行”（即数据库中有多少个数据行符合这条UPDATE语句的选取条件），而不是它“实际修改了多少个数据行”。如果应用程序要用到这个含义，就需要在连接MySQL服务器的时候向mysql\_real\_connect()函数的flags参数传递一个CLIENT\_FOUND\_ROWS值。

### 6.5.2 处理有结果集的查询

有些查询是会返回结果集的。在用mysql\_query()或mysql\_real\_query()发出这类查询命令之后，它们从数据库里检索出来的数据将被返回为一个结果集供你做进一步的处理。注意，在MySQL里，能返回结果集的语句并非仅有SELECT这一条，SHOW、DESCRIBE、EXPLAIN、CHECK TABLE等语句也会返回结果集。对于以上这些语句，在发出查询命令后通常还需要对它们返回的结果集做进一步的处理。

结果集的处理工作涉及以下几个步骤：

1) 通过调用mysql\_store\_result()或mysql\_use\_result()函数生成结果集。这两个函数在调用成功时都将返回一个MYSQL\_RES指针；如果执行失败，则都将返回NULL。我们稍后将会对这两个函数之间的区别和它们各自的适用场合进行介绍。但就眼下来说，我们将以mysql\_store\_result()为例展开讨论，这个函数会立刻从MySQL服务器检索出有关的数据行并把它们保存在客户主机上。

2) 调用mysql\_fetch\_row()函数依次取回结果集里的各个数据行。这个函数在调用成功时将返回一个MYSQL\_ROW值；如果已经到达结果集里的最后一个数据行，则将返回NULL。MYSQL\_ROW值其实是一个字符串数组指针，数组中的字符串代表着数据行中的各个数据列的值。如何处理这些数据行要由应用程序的用途来决定。可以简单地把它们都打印出来，可以对它们进行统计学分析，或者做一些其他的处理。

3) 完成结果集的处理工作之后，调用mysql\_free\_result()函数释放所占用的内存资源。如果忘了做这件事，应用程序就会造成内存泄漏。如果应用程序的运行时间比较长，就要特别注意及时释放那些不再使用的结果集；否则，系统会因资源消耗量持续增长而变得越来越慢。

下面是一个用来对有结果集的查询进行处理的代码框架：

```
MYSQL_RES *res_set;

if (mysql_query (conn, "SHOW TABLES FROM sampdb") != 0)
    print_error (conn, "mysql_query() failed");
else
{
    res_set = mysql_store_result (conn);    /* generate result set */
    if (res_set == NULL)
        print_error (conn, "mysql_store_result() failed");
    else
    {
        /* process result set, then deallocate it */
        process_result_set (conn, res_set);
        mysql_free_result (res_set);
    }
}
```

这段代码把结果集的处理细节都隐藏在了process\_result\_set()函数里，而这一小节里的目标就是对这个函数进行定义。一般来说，结果集的处理工作都是以一个如下所示的循环为基础的：



```

MYSQL_ROW row;

while ((row = mysql_fetch_row (res_set)) != NULL)
{
    /* do something with row contents */
}

```

mysql\_fetch\_row()将返回一个MYSQL\_ROW值，它是一个数组指针。如果把这个返回值赋值给一个名为row的变量，就可以利用row[i]语法来访问其中的各个元素，i的取值范围是0到该数据行里的数据列个数减去1。MYSQL\_ROW数据类型有以下几个值得注意的要点：

- MYSQL\_ROW本身是一个指针类型，所以在定义这种类型的变量时应该把它写成“MYSQL\_ROW row”而不能写成“MYSQL\_ROW \*row”。
- 在MYSQL\_ROW数组里，所有数据类型（包括数值型）都将被返回为字符串。如果想把某个值视为数字，就必须亲自对之进行类型转换。
- MYSQL\_ROW数组里的字符串都是以NULL字节结尾的。但是，因为那些用来存放二进制数据的数据列里可能包含着NULL字节本身，所以不能把这个数组里的值视为以NULL字节结尾的字符串。必须通过数据列的长度来了解它们到底有多长。（数据列长度的确定办法将在第6.5.6节介绍。）
- 在MYSQL\_ROW数组里，数据库里的NULL值将被表示为一个NULL指针。如果数据库中的某个数据列没有被定义为NOT NULL，就必须在程序里检查来自该数据列里的数据值是否为NULL，否则，程序就会因试图对NULL指针进行求值而崩溃。

如何处理结果集里的数据行要由应用程序的具体用途来决定。作为演示，这里的示例程序将只把结果集里的数据行打印出来，数据列值之间用制表符隔开。要想做到这一点，就必须知道数据行是由多少个数据列构成的，这个信息可以用另一个MySQL C API函数mysql\_num\_fields()查出来。

下面是process\_result\_set()函数的源代码：

```

void
process_result_set (MYSQL *conn, MYSQL_RES *res_set)
{
    MYSQL_ROW      row;
    unsigned int    i;

    while ((row = mysql_fetch_row (res_set)) != NULL)
    {
        for (i = 0; i < mysql_num_fields (res_set); i++)
        {
            if (i > 0)
                fputc ('\t', stdout);
            printf ("%s", row[i] != NULL ? row[i] : "NULL");
        }
        fputc ('\n', stdout);
    }
}

```

```

    }
    if (mysql_errno (conn) != 0)
        print_error (conn, "mysql_fetch_row() failed");
    else
        printf ("%lu rows returned\n",
            (unsigned long) mysql_num_rows (res_set));
}

```

这个process\_result\_set()函数将依次打印各数据行的内容，数据列值之间用制表符分隔（NULL值将被打印为单词“NULL”）；最后再把一个表明结果集里有多少个数据行的计数值打印出来，这个计数值是通过调用mysql\_num\_rows()函数而得到的。类似于mysql\_affected\_rows()，mysql\_num\_rows()函数的返回值也是一个my\_ulonglong值，所以得先把它转换为unsigned long类型，再用“%lu”格式符去打印它。但同时也要注意：mysql\_affected\_rows()函数的输入参数是一个连接句柄，而mysql\_num\_rows()函数的输入参数却是一个结果集指针。

负责打印数据行计数值的代码是一个条件语句，它先要判断是否有错误发生。做出这种预防性安排的理由是：如果结果集是用mysql\_store\_result()创建的，那么mysql\_fetch\_row()的NULL返回值将永远意味着“已经到达结果集的末尾”；可如果结果集是用mysql\_use\_result()创建的，那么mysql\_fetch\_row()的NULL返回值就有“已经到达结果集的末尾”和“发生错误”两种含义。因为process\_result\_set()并不知道自己的父函数是用mysql\_store\_result()还是用mysql\_use\_result()去生成结果集的，所以，为了让它在这两种情况下都能正确地检测到出错情况，我们给它加上了这个出错情况测试。

在打印数据列值的时候，process\_result\_set()函数的这一版本采用了最简单因而也最粗糙的做法。比如说，假设执行的是如下所示的查询：

```

SELECT last_name, first_name, city, state FROM president
ORDER BY last_name, first_name

```

将看到下面这样的输出，它可算不上整齐：

```

Adams    John    Braintree  MA
Adams    John Quincy Braintree  MA
Arthur   Chester A.  Fairfield  VT
Buchanan James    Mercersburg PA
Bush     George H.W. Milton    MA
Bush     George W.   New Haven  CT
Carter   James E.    Plains    GA
...

```

如果能给这份输出加上数据列名称作为列标题，再把数据沿纵向对齐，效果就会好得多。这就需要我们知道各数据列的名字和各数据列里最宽的值。这些信息是存在的，但它们与数据列的数据值并没有放在一起——它们是结果集的元数据（即关于数据的数据）的组成部分。等把查询处理程序做得更完善一些之后，我们将在第6.5.6节编写一个更整齐美观的排版打印模块。

### 如何打印二进制数据

包含二进制数据的数据列值可能包含NULL字节，这种数据是无法用printf()函数的“%s”格式符来打印的。printf()函数把NULL字节视为字符串的结束标记，所以它只能把数据列值中的第一个NULL字节之前的内容打印出来。因此，要想把二进制数据完整地打印出来，就必须使用根据数据本身的长度来打印的工具，比如fwrite()函数。

### 6.5.3 一个通用的查询处理程序

前面的查询处理示例都是根据SQL语句是否会返回一个结果集而决定如何进行处理。这种做法能够奏效的原因是我们把查询命令都硬编码在了程序代码里：在示例中分别使用了一条不会返回结果集的INSERT语句和一条会返回结果集的SHOW TABLES语句，并且都顺利地完成了对它们的处理。

可是，这种事先知道将要处理哪一种查询的好事不会总让你遇到。比如说，如果客户程序将要执行的查询命令是从键盘或者某个文件读入的，那么，不仅很难事先知道它的具体内容，就连它是否是一条合法的SQL语句都成问题。你该怎么办？虽说有一定的可行性，但你肯定不想通过词法、语法分析来确定它们到底是哪一种SQL语句。而且，这个方案也不像它乍看上去那么简单：如果只检查语句的第一个单词是不是“SELECT”，就无法对付下面这种以注释开头的合法语句：

```
/* comment */ SELECT ...
```

还好，有了MySQL C客户程序开发库的帮忙，不必提前知道数据库查询命令的具体类型就能对它做出正确的处理。下面，我们将利用这个函数库来编写一个通用的查询处理程序，它能对各种SQL语句（不管它是否会返回一个结果集，也不管它的执行是否成功）做出正确的处理。在开始编写这个处理程序的代码之前，先介绍一下它的工作流程：

- 1) 发出数据库查询命令。如果执行失败，则就此结束。
- 2) 如果查询成功，调用mysql\_store\_result()函数从服务器检索出有关的数据行并创建一个结果集。
- 3) 如果mysql\_store\_result()调用成功，查询将返回一个结果集。通过调用mysql\_fetch\_row()函数对结果集里的数据行进行处理直到它返回NULL为止，然后释放这个结果集。
- 4) 如果mysql\_store\_result()调用失败，其原因可能是：1) 这个查询根本不会返回一个结果集；2) 这个查询会返回一个结果集，但在试图检索结果集时发生了错误。这两种情况可以利用mysql\_field\_count()函数来区别，把连接句柄传递给这个函数，然后检查它的返回值：
  - 如果mysql\_field\_count()返回的是0，说明这个查询一个数据列也没有返回，也就是没有结果集。（这同时也表明查询命令是INSERT、DELETE、UPDATE等语句。）
  - 如果mysql\_field\_count()返回的是一个非零值，说明这个查询应该返回一个结果集，因为现在不是这样，肯定是发生了错误。发生这类错误的原因有很多，比如因结果集过大而导致内存分配失败，或者客户和服务端之间的网络连接出现故障等等。

下面这个函数能够对任意查询进行处理，它的输入参数有两个：一个是连接句柄，另一个是以NULL字节结尾的查询命令字符串。

```
void
process_query (MYSQL *conn, char *query)
{
    MYSQL_RES *res_set;
    unsigned int field_count;

    if (mysql_query (conn, query) != 0) /* the query failed */
    {
        print_error (conn, "Could not execute query");
        return;
    }

    /* the query succeeded; determine whether or not it returns data */

    res_set = mysql_store_result (conn);
    if (res_set) /* a result set was returned */
    {
        /* process rows, then free the result set */
        process_result_set (conn, res_set);
        mysql_free_result (res_set);
    }
    else /* no result set was returned */
    {
        /*
         * does the lack of a result set mean that the query didn't
         * return one, or that it should have but an error occurred?
         */
        if (mysql_field_count (conn) == 0)
        {
            /*
             * query generated no result set (it was not a SELECT, SHOW,
             * DESCRIBE, etc.), so just report number of rows affected
             */
            printf ("%lu rows affected\n",
                    (unsigned long) mysql_affected_rows (conn));
        }
        else /* an error occurred */
        {
            print_error (conn, "Could not retrieve result set");
        }
    }
}
```

不过，这里还有一个小问题要解决：MySQL 3.22.24之前的版本没有提供mysql\_field\_count()函数。这个小缺陷可以这样来解决：在MySQL 3.22.24之前的版本里，用mysql\_num\_fields()来

代替mysql\_field\_count()。如果想让编写的客户程序在MySQL软件的任何版本下都能工作,请把下面这个代码段添加到程序的源文件里去——这段代码应该放在mysql.h指令之后、调用mysql\_field\_count()之前:

```
#if !defined(MYSQL_VERSION_ID) || (MYSQL_VERSION_ID<32224)
#define mysql_field_count mysql_num_fields
#endif
```

这样,如果MySQL软件版本低于3.22.24,上面的代码段就将把mysql\_field\_count()调用全部转换为mysql\_num\_fields()调用。

#### 6.5.4 另一种查询处理方案

上一小节里的process\_query()函数有以下三个特点:

- 它使用一个以NULL字节结尾的字符串和mysql\_query()来发出查询命令。
- 它使用mysql\_store\_result()来检索结果集。
- 在没有获得结果集的时候,它利用mysql\_field\_count()来判断其原因是查询命令执行出错还是它根本就不会返回结果集。

如果在这三个方面加以改变,就能得到另一种查询处理方案。我们可以:

- 用一个计数查询字符串和mysql\_real\_query()代替以NULL字节结尾的字符串和mysql\_query()。
- 用mysql\_use\_result()代替mysql\_store\_result()创建结果集。
- 用mysql\_error()或mysql\_errno()代替mysql\_field\_count()去区分“执行出错”和“没有结果集可供返回”这两种情况。

可以按这三种情况的任意组合对process\_query()函数进行修改。下面这个process\_real\_query()函数就是对process\_query()函数的上述三个方面全部进行替换后得到的:

```
void
process_real_query (MYSQL *conn, char *query, unsigned int len)
{
    MYSQL_RES *res_set;
    unsigned int field_count;

    if (mysql_real_query (conn, query, len) != 0)    /* the query failed */
    {
        print_error (conn, "Could not execute query");
        return;
    }

    /* the query succeeded; determine whether or not it returns data */

    res_set = mysql_use_result (conn);
    if (res_set)    /* a result set was returned */
    {
```



```

        /* process rows, then free the result set */
        process_result_set (conn, res_set);
        mysql_free_result (res_set);
    }
    else /* no result set was returned */
    {
        /*
         * does the lack of a result set mean that the query didn't
         * return one, or that it should have but an error occurred?
         */
        if (mysql_errno (conn) == 0)
        {
            /*
             * query generated no result set (it was not a SELECT, SHOW,
             * DESCRIBE, etc.), so just report number of rows affected
             */
            printf ("%lu rows affected\n",
                    (unsigned long) mysql_affected_rows (conn));
        }
        else /* an error occurred */
        {
            print_error (conn, "Could not retrieve result set");
        }
    }
}

```

#### 6.5.5 mysql\_store\_result()与mysql\_use\_result()函数的对比

mysql\_store\_result()与mysql\_use\_result()函数的共同点有两个：一是都以一个连接句柄作为输入参数，二是都会返回一个结果集。它们之间的差异却远多于此。这两个函数最本质的区别在于它们用来从服务器检索结果集数据行的方式不一样：mysql\_store\_result()会在你调用它时立刻把所有的数据行全都检索出来并保存到客户主机上；mysql\_use\_result()则只完成对结果集的初始化工作，它本身并不取回任何数据行。正是这一区别导致了这两个函数在其他方面的种种差异。为了让大家能够在编写MySQL客户程序的时候在这两个函数之间做出最适当的选择，本小节将对这两个函数进行全面的比较。

在从服务器检索数据行的时候，mysql\_store\_result()会取回所有的数据行，为它们分配内存，然后把它们保存在客户主机里。此后的mysql\_fetch\_row()调用永远不会返回一个出错，因为它们只是简单地从一个已经存在的数据结构里提取出一个数据行而已。因此，mysql\_fetch\_row()函数的NULL返回值永远意味着“已经到达结果集的末尾”。

再看mysql\_use\_result()，它本身不会去检索任何数据行。它只完成对结果集的初始化工作并表明客户程序将依次取回各有关数据行，真正取回各数据行的工作还要由你通过一系列的mysql\_fetch\_row()调用去完成。因此，虽然mysql\_fetch\_row()调用的NULL返回值在大多数情况下仍然意味着“已经到达结果集的末尾”，但也有可能是表明“与服务器的通信出现了问题”。

可以利用mysql\_error()或mysql\_errno()调用来区分这两种情况。

因为要把结果集完整地保存在客户机里，所以，与mysql\_use\_result()相比，mysql\_store\_result()消耗的内存和其他系统资源更多，因分配内存和创建各种必要的数据结构而导致的开销也更大。过大的结果集会给客户主机带来内存消耗殆尽的风险。因此，当觉得某个查询命令会生成一个包含有许多数据行的结果集时，就应该使用mysql\_use\_result()。

再看mysql\_use\_result()，因为每次只取回一个数据行进行处理，所以它对内存的要求很低。同时，因为不必为创建结果集而建立各种复杂的数据结构，它的内存分配工作也将完成得更快。但从另一个角度看，mysql\_use\_result()加重了服务器的负担，服务器必须把结果集里的数据行一直保存到客户程序把它们都检索到客户机为止。因此，mysql\_use\_result()不适合用在以下几种客户程序里：

- 根据用户请求逐个遍历各有关数据行的交互式客户程序。（你肯定不想让服务器因为用户去喝咖啡了而一直等着发送下一个数据行，对吧？）
- 在前、后两次数据行检索操作之间需要进行大量处理的客户程序。

在上述两种情况里，客户程序都无法迅速地把结果集里的数据行全部检索完毕。这对服务器和其他客户程序都有很大的负面影响，因为从中检索数据的那些数据表在结束这次查询之前将一直处于读操作锁定状态，试图修改这些数据表或者试图往里面插入新记录的其他客户程序都将因此而被阻塞。

虽说mysql\_store\_result()会消耗较多的内存，但让你能对整个结果集进行访问却是一件好事。因为结果集里的数据行都存放在客户机里，所以可以以随机方式去访问它们，可以利用mysql\_data\_seek()、mysql\_row\_seek()、mysql\_row\_tell()等函数按任意顺序去访问数据行。可如果当初使用的是mysql\_use\_result()的话，就只能对mysql\_fetch\_row()取回的当前数据行进行了。如果想按任意顺序而不是按它们从服务器被依次取回的顺序去处理结果集里的数据行，就必须使用mysql\_store\_result()。比如说，如果想让编写的应用程序允许用户跳跃地前、后浏览用某个查询选取出来的数据行，mysql\_store\_result()就应该是首选。

mysql\_store\_result()还能让你访问到一些使用mysql\_use\_result()访问不到的数据列信息。比如说，可以通过调用mysql\_num\_rows()查知结果集总共包含多少个数据行；可以从MYSQL\_FIELD数据列信息结构的max\_width成员查知各数据列的数据最大宽度。可如果当初使用的是mysql\_use\_result()的话，mysql\_num\_rows()将只有在数据行全部取回之后才会返回正确的计数值；类似地，max\_width成员的值也只有在数据行全部取回之后才能正确地计算出来，在此之前将不可用。

因为mysql\_use\_result()做的事情比mysql\_store\_result()少，所以它必须遵守一条mysql\_store\_result()不必遵守的限制性规定——客户程序必须通过调用mysql\_fetch\_row()取回结果集里的每一个数据行。如果在发出另一条查询之前忘了这么做，当前结果集里尚未来得及取回的数据行就将混杂在下一个查询的结果集里，而你则会看到一条“out of sync”（数据不同步）出错信息。（如果在发出第二个查询之前调用了mysql\_free\_result()函数，就可以避免出现这一问题。mysql\_free\_result()将取回并丢弃当前结果集里尚未被取回的数据行。）这条限制性规定还隐含着这样一层含义：如果使用的是mysql\_use\_result()，那每次只能使用一个结果集进行工作。

`mysql_store_result()`不会发生数据不同步的问题,这是因为,当这个函数返回的时候,服务器上就不会再有尚未被取回的数据行了。事实上,如果使用的是`mysql_store_result()`,根本用不着采用调用`mysql_fetch_row()`函数的办法去取回数据行。不过,如果感兴趣的只是结果集是否为空而不是结果集里有什么样的数据,这个函数还是有些用处的。比如说,想知道数据库里是否存在着一个名为`mytbl`的数据表,于是发出了一个如下所示的查询:

```
SHOW TABLES LIKE 'mytbl'
```

如果在调用了`mysql_store_result()`之后,`mysql_num_rows()`返回的是一个非零值,就说明这个数据表是存在的。就这个例子而言,虽然`mysql_fetch_row()`也可以告诉你`mytbl`是否存在,但并不是非得调用它不可。

虽说应该及时调用`mysql_free_result()`函数去释放在`mysql_store_result()`生成的结果集,但并非必须在发出下一个查询之前这样做。这意味着可以同时生成并使用多个结果集进行工作,这与`mysql_use_result()`要求每次只能使用一个结果集进行工作的限制性规定形成了鲜明的对照。

如果想向用户提供最大限度的灵活性,可以让用户去选择结果集的处理方案。MySQL软件自带的`mysql`和`mysqldump`就是两个很好的例子:在默认的情况下,它们将使用`mysql_store_result()`去生成结果集;可如果在命令行上给出了`--quick`选项,它们就将切换使用`mysql_use_result()`。

#### 6.5.6 结果集元数据的使用

结果集并不是仅包含着从数据库里检索出来的数据,它还包含着关于这些数据的描述性信息,即所谓的“结果集元数据”。结果集元数据包括以下一些信息:

- 结果集里的数据行个数和数据列个数。只需调用`mysql_num_rows()`和`mysql_num_fields()`函数就能把它们查出来。
- 当前数据行里各数据列值的长度。只需调用`mysql_fetch_lengths()`函数就能把它们查出来。
- 关于各数据列的描述性信息,比如数据列的名字和类型、各数据列的数据最大宽度、结果集里的数据列都是从数据库的哪些数据表里检索出来的等等。这些信息都保存在与各数据列相对应的`MYSQL_FIELD`结构里,可以通过调用`mysql_fetch_field()`函数去获取它们。本书的附录F对`MYSQL_FIELD`结构以及各种用来访问这些数据列描述信息的API函数都做了比较详细的介绍。

数据列元数据是否存在或是否有效还部分地取决于所采用的结果集处理方案。正如第6.5.5节里论述的那样,如果要用到数据行计数值或者各数据列的数据最大宽度,就必须使用`mysql_store_result()`而不是`mysql_use_result()`去创建结果集。

结果集元数据有助于决定如何对结果集数据进行处理:

- 如果想生成一份带有列标题并沿纵向整齐排列的输出报告,就需要用到各数据列的名字和它们的数据最大宽度。
- 在需要依次对各数据行中的各个数据列进行某种处理的场合,可以把数据行计数值和数据列计数值用做循环控制变量。

- 在需要根据结果集中的数据行和数据列个数来为某种数据结构分配内存的场合，可能需要用到数据行计数值和数据列计数值。
- 可以利用结果集元数据来确定数据列的数据类型。可以清楚地掌握哪些数据列是数值类型的、哪些又是字符串类型的、它们是否包含着二进制数据，等等。

在第6.5.2节里，所编写的`process_result_set()`函数将把结果集里的数据行按各数据列以制表符分隔的格式打印出来。这个程序很有用（比如说，可以用它把数据导入到一个电子表格里），但它的显示格式却不怎么整齐，既不容易发现其中的打字错误，也不适合作为正式的报告。下面就是那个`process_result_set()`函数生成的输出报告：

```
Adams   John   Braintree   MA
Adams   John Quincy Braintree   MA
Arthur  Chester A. Fairfield   VT
Buchanan James  Mercersburg PA
Bush    George H.W. Milton    MA
Bush    George W.   New Haven   CT
Carter  James E.    Plains     GA
...
```

下面，我们将改写`process_result_set()`函数，使它能够生成一份表格化的输出报告，把各数据列分别放在一个框里并给它们加上标题。下面就是新`process_result_set()`函数生成的输出报告：

```
+-----+-----+-----+-----+
| last_name | first_name | city | state |
+-----+-----+-----+-----+
| Adams | John | Braintree | MA |
| Adams | John Quincy | Braintree | MA |
| Arthur | Chester A. | Fairfield | VT |
| Buchanan | James | Mercersburg | PA |
| Bush | George H.W. | Milton | MA |
| Bush | George W. | New Haven | CT |
| Carter | James E. | Plains | GA |
...
```

新`process_result_set()`函数的打印输出部分将按顺序做以下几件事：

- 1) 确定各数据列的显示宽度。
- 2) 打印标题行，列标题之间用垂直线字符分隔，整个标题行放在上、下两行短划线之间。
- 3) 把结果集里的数据行依次打印出来，数据列之间用垂直线字符分隔并沿纵向对齐。如果是数字，则按居右方式打印；如果是NULL值，则打印为单词“NULL”。
- 4) 最后，把结果集里的数据行总数打印在表格的下面。

这个练习很好地演示了结果集元数据的用法，要想完成这一任务，除来自MySQL数据库的数据外，还需要知道并使用关于结果集本身的很多信息。

有人也许会这样想：“这好像与mysql程序生成的输出报告差不多。”是的，的确如此。希望大家把mysql程序的源代码与`process_result_set()`函数的最终代码做一下对比。它们并不相同，



但这种对比肯定会给大家带来很多启发。

首先，我们要为各数据列确定一个显示宽度。这项工作将由如下所示的代码负责完成。请注意：这段代码里的各种计算全部是根据结果集元数据进行的，没有涉及任何来自MySQL数据库的数据。

```
MYSQL_FIELD      *field;
unsigned long     col_len;
unsigned int      i;

/* determine column display widths -- requires result set to be */
/* generated with mysql_store_result(), not mysql_use_result() */
mysql_field_seek (res_set, 0);
for (i = 0; i < mysql_num_fields (res_set); i++)
{
    field = mysql_fetch_field (res_set);
    col_len = strlen (field->name);
    if (col_len < field->max_length)
        col_len = field->max_length;
    if (col_len < 4 && !IS_NOT_NULL (field->flags))
        col_len = 4;      /* 4 = length of the word "NULL" */
    field->max_length = col_len; /* reset column info */
}
```

为了计算出各数据列的显示宽度，这段代码将遍历与结果集里的各个数据列相对应的MYSQL\_FIELD结构。在进入循环之前，我们先通过调用mysql\_field\_seek()定位到第一个MYSQL\_FIELD结构处；在进入循环之后，再通过调用mysql\_fetch\_field()返回指向与数据列相对应的MYSQL\_FIELD结构的指针。数据列的显示宽度是下面这三个值中的最大值，这三个值全部是根据MYSQL\_FIELD结构里的元数据而获得的：

- field->name的长度，即数据列标题的长度。
- field->max\_length，数据列中最长的那个数据值的长度。
- 字符串“NULL”的长度（如果数据列允许包含NULL值的话）。将根据field->flags来判断数据列是否允许包含NULL值。

请注意：在把数据列的显示宽度确定下来之后，我们把它赋值给了MYSQL\_FIELD结构中的max\_length成员。可是，MYSQL\_FIELD结构是从MySQL C客户程序开发库那里获得的，它是否允许修改呢？换个问法，MYSQL\_FIELD结构的内容是不是只读的呢？我得承认，我认为它是只读的。但在MySQL软件自带的客户程序中，有一些也像我们这样对max\_length进行过修改，所以我认为这样做也算合法。（如果不想修改max\_length成员，可以分配一个unsigned long数组来存放计算出来的显示宽度。）

显示宽度的计算工作还有一个地方需要特别注意。大家知道，如果结果集是用mysql\_use\_result()生成的，max\_length将毫无意义。因为现在是用max\_length来确定数据列显示宽度的，所以这里需要假设process\_result\_set()函数将被用在一个用mysql\_store\_result()来生成结果集的程序里。如果想把process\_result\_set()函数用在一个使用mysql\_use\_result()而不是



mysql\_store\_result()来生成结果集的程序里,就必须改变这里的做法,可以使用MYSQL\_FIELD结构中的length成员,它给出的是数据列值能够达到的最大长度。

把数据列的显示宽度确定下来之后,就可以开始打印了。列标题的问题很容易解决:对于给定的数据列,只需把与该数据列对应的MYSQL\_FIELD结构中的name成员打印出来就行了,它的打印宽度就是刚才计算出来的该数据列的显示宽度:

```
printf (" %-*s |", (int) field->max_length, field->name);
```

至于来自MySQL数据库的数据,我们将使用一个循环来遍历结果集里的各个数据行,在每次循环中把当前数据行的数据列值依次打印出来。数据列值的打印工作也有一个地方需要注意,因为它可能是一个NULL值或者是一个数值(数值必须按居右格式打印)。下面是用来打印数据列值的代码,其中, row[i]存放着当前数据行,指针field则指向当前数据列的MYSQL\_FIELD结构:

```
if (row[i] == NULL)                /* print the word "NULL" */
    printf (" %-*s |", (int) field->max_length, "NULL");
else if (IS_NUM (field->type)) /* print value right-justified */
    printf (" %-*s |", (int) field->max_length, row[i]);
else                               /* print value left-justified */
    printf (" %-*s |", (int) field->max_length, row[i]);
```

如果某个数据列的field->type表明它是一个数值类型的数据列(例如, INT、FLOAT、DECIMAL等等), IS\_NUM()宏的求值结果将为真。

用来打印结果集数据的最终代码如下所示。因为将多次打印由短划线字符构成的表格线,所以我们还编写了一个print\_dashes()函数来做这件事,这要比在多个地方重复使用同样的代码来打印这些表格线的做法好。

```
void
print_dashes (MYSQL_RES *res_set)
{
    MYSQL_FIELD      *field;
    unsigned int      i, j;

    mysql_field_seek (res_set, 0);
    fputc ('+', stdout);
    for (i = 0; i < mysql_num_fields (res_set); i++)
    {
        field = mysql_fetch_field (res_set);
        for (j = 0; j < field->max_length + 2; j++)
            fputc ('-', stdout);
        fputc ('+', stdout);
    }
    fputc ('\n', stdout);
}

void
process_result_set (MYSQL *conn, MYSQL_RES *res_set)
```

```

{
MYSQL_ROW      row;
MYSQL_FIELD    *field;
unsigned long   col_len;
unsigned int    i;

/* determine column display widths -- requires result set to be */
/* generated with mysql_store_result(), not mysql_use_result() */
mysql_field_seek (res_set, 0);
for (i = 0; i < mysql_num_fields (res_set); i++)
{
    field = mysql_fetch_field (res_set);
    col_len = strlen (field->name);
    if (col_len < field->max_length)
        col_len = field->max_length;
    if (col_len < 4 && !IS_NOT_NULL (field->flags))
        col_len = 4; /* 4 = length of the word "NULL" */
    field->max_length = col_len; /* reset column info */
}

print_dashes (res_set);
fputc ('|', stdout);
mysql_field_seek (res_set, 0);
for (i = 0; i < mysql_num_fields (res_set); i++)
{
    field = mysql_fetch_field (res_set);
    printf (" %-*s |", (int) field->max_length, field->name);
}
fputc ('\n', stdout);
print_dashes (res_set);

while ((row = mysql_fetch_row (res_set)) != NULL)
{
    mysql_field_seek (res_set, 0);
    fputc ('|', stdout);
    for (i = 0; i < mysql_num_fields (res_set); i++)
    {
        field = mysql_fetch_field (res_set);
        if (row[i] == NULL) /* print the word "NULL" */
            printf (" %-*s |", (int) field->max_length, "NULL");
        else if (IS_NUM (field->type)) /* print value right-justified */
            printf (" %*s |", (int) field->max_length, row[i]);
        else /* print value left-justified */
            printf (" %-*s |", (int) field->max_length, row[i]);
    }
    fputc ('\n', stdout);
}
print_dashes (res_set);
printf ("%lu rows returned\n", (unsigned long) mysql_num_rows (res_set));
}

```

MySQL C 客户程序开发库还提供了其他一些API函数用来访问MYSQL\_FIELD结构。比如说，在上面的代码里，我们曾多次使用下面这样的循环语句来访问MYSQL\_FIELD结构：

```
mysql_field_seek (res_set, 0);
for (i = 0; i < mysql_num_fields (res_set); i++)
{
    field = mysql_fetch_field (res_set);
    ...
}
```

但mysql\_field\_seek()和mysql\_fetch\_field()的组合并非访问MYSQL\_FIELD结构的惟一手段。本书的附录F还介绍了mysql\_fetch\_fields()和mysql\_fetch\_field\_direct()等其他几个用来访问MYSQL\_FIELD结构的API函数。

## 6.6 客户程序4——交互式查询程序

在这一节里，将把此前开发的代码结合起来去编写一个简单的交互式客户程序client4。这个程序将接收输入的查询命令、用通用查询处理函数process\_query()执行它们，再用前面开发的排版输出函数process\_result\_set()把查询结果打印出来。

client4程序与MySQL软件自带的mysql客户程序很相似，当然在功能上没有那么丰富。client4程序对输入做了几项限制：

- 每个输入行只能包含一条完整的语句。
- 语句不需要以分号(;)或者“\g”结尾。
- 只支持一条非SQL命令，即用来结束程序运行的“quit”和“\q”。还可以使用Ctrl-D组合键来退出运行。

有了前面那些成果，client4程序就很容易写了（新代码大概只有十几行）。客户程序框架（client3.c）和我们已经完成的那几个工具函数几乎提供了所有的东西，只需再增加一个用来接收输入行并执行它们的循环就大功告成了。

编写client4程序的第一步是用客户程序框架client3.c拷贝一份client4.c源文件，然后再把process\_query()、process\_result\_set()和print\_dashes()函数添加到client4.c源文件里面去。最后，在client4.c源文件里，在main()函数中找到如下所示的那一行：

```
/* ... issue queries and process results here ... */
```

把这一行替换为如下所示的while循环：

```
while (1)
{
    char    buf[10000];

    fprintf (stderr, "query> ");                /* print prompt */
    if (fgets (buf, sizeof (buf), stdin) == NULL) /* read query */
        break;
    if (strcmp (buf, "quit\n") == 0 || strcmp (buf, "\\q\n") == 0)
        break;
    process_query (conn, buf);                    /* execute query */
}
```

现在,把源文件client4.c编译为目标文件client4.o,再把client4.o与MySQL C客户程序开发库链接起来生成可执行文件client4,我们就将得到一个能够执行任何查询并显示其结果的交互式MySQL客户程序。下面几个例子演示了client4程序的工作情况,我们试用了一个SELECT查询和一个非SELECT查询,还故意给出了几个有错误的语句:

```
% ./client4
query> USE sampdb
0 rows affected
query> SELECT DATABASE(), USER()
+-----+-----+
| DATABASE() | USER() |
+-----+-----+
| sampdb     | sampadm@localhost |
+-----+-----+
1 rows returned
query> SELECT COUNT(*) FROM president
+-----+
| COUNT(*) |
+-----+
|         42 |
+-----+
1 rows returned
query> SELECT last_name, first_name FROM president ORDER BY last_name LIMIT 3
+-----+-----+
| last_name | first_name |
+-----+-----+
| Adams    | John      |
| Adams    | John Quincy |
| Arthur   | Chester A. |
+-----+-----+
3 rows returned
query> CREATE TABLE t (i INT)
0 rows affected
query> SELECT j FROM t
Could not execute query
Error 1054 (Unknown column 'j' in 'field list')
query> USE mysql
Could not execute query
Error 1044 (Access denied for user: 'sampadm@localhost' to database 'mysql')
```

## 6.7 编写具备SSL支持的客户程序

MySQL 4新增了一些SSL支持功能,在你自己的程序里,可以利用这些功能通过安全化连接去访问MySQL服务器。在这一节里,我们将在client4程序的基础上编写一个用途相似的sslclient程序,这两个程序几乎一模一样,只是后者多了一项建立加密连接的能力。如果想使用sslclient程序去建立安全化连接,就必须满足两个条件:一是MySQL软件必须编译有SSL支持组

件;二是MySQL服务器在启动时已经通过适当的选项把自己的数字证书文件和密钥文件设定好了。此外,还需要把客户端的数字证书文件和密钥文件准备好。这些工作的具体做法请参考本书第12.3节。建议大家在MySQL 4.0.5及以后的版本下使用sslclient程序,因为较早的MySQL 4.0.x版本中的SSL和选项处理例程与这里的描述略有差异。

这个程序的源代码文件sslclient.c已经收录在sampdb发行版本里了,可以直接用它建立sslclient客户程序。从client4.c文件开始创建sslclient.c文件的过程如下所示:

1) 用client4.c复制一份sslclient.c文件。以下修改都将在sslclient.c文件里进行。

2) 为了让编译器检测出SSL支持是否可用,MySQL在头文件my\_config.h里相应地定义了一个名为HAVE\_OPENSSL的符号。在编写与SSL有关的代码时,应该使用如下所示的构造;这样,如果无法使用SSL,与SSL有关的代码将被忽略。

```
#ifdef HAVE_OPENSSL
    ...在这里写出与SSL有关的代码...
#endif
```

头文件my\_config.h可以通过头文件my\_global.h被包括到程序源文件里。因为sslclient.c文件已经把头文件my\_global.h包括了进来,所以就不必再明确地包括my\_config.h头文件了。

3) 修改以my\_option选项信息结构为元素的my\_opts数组,把与SSL有关的标准选项(--ssl-ca、--ssl-key等等)也添加进去。最简便的办法是用一条#include指令把sslopt-longopts.h头文件的内容包括到my\_opts数组里来。修改后的my\_opts数组如下所示:

```
static struct my_option my_opts[] = /* option information structures */
{
    {"help", '?', "Display this help and exit",
     NULL, NULL, NULL,
     GET_NO_ARG, NO_ARG, 0, 0, 0, 0, 0, 0},
    {"host", 'h', "Host to connect to",
     (gptr *) &opt_host_name, NULL, NULL,
     GET_STR_ALLOC, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
    {"password", 'p', "Password",
     (gptr *) &opt_password, NULL, NULL,
     GET_STR_ALLOC, OPT_ARG, 0, 0, 0, 0, 0, 0},
    {"port", 'P', "Port number",
     (gptr *) &opt_port_num, NULL, NULL,
     GET_UINT, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
    {"socket", 'S', "Socket path",
     (gptr *) &opt_socket_name, NULL, NULL,
     GET_STR_ALLOC, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
    {"user", 'u', "User name",
     (gptr *) &opt_user_name, NULL, NULL,
     GET_STR_ALLOC, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
    #include <sslopt-longopts.h>

    { NULL, 0, NULL, NULL, NULL, NULL, GET_NO_ARG, NO_ARG, 0, 0, 0, 0, 0, 0 }
};
```



sslopt-longopts.h是一个MySQL公共头文件。它的内容如下所示（格式稍做调整）：

```
#ifndef HAVE_OPENSSL
    {"ssl", OPT_SSL_SSL,
     "Enable SSL for connection. Disable with --skip-ssl",
     (gp_ptr*) &opt_use_ssl, NULL, 0,
     GET_BOOL, NO_ARG, 0, 0, 0, 0, 0, 0},
    {"ssl-key", OPT_SSL_KEY, "X509 key in PEM format (implies --ssl)",
     (gp_ptr*) &opt_ssl_key, NULL, 0,
     GET_STR, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
    {"ssl-cert", OPT_SSL_CERT, "X509 cert in PEM format (implies --ssl)",
     (gp_ptr*) &opt_ssl_cert, NULL, 0,
     GET_STR, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
    {"ssl-ca", OPT_SSL_CA,
     "CA file in PEM format (check OpenSSL docs, implies --ssl)",
     (gp_ptr*) &opt_ssl_ca, NULL, 0,
     GET_STR, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
    {"ssl-capath", OPT_SSL_CAPATH,
     "CA directory (check OpenSSL docs, implies --ssl)",
     (gp_ptr*) &opt_ssl_capath, NULL, 0,
     GET_STR, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
    {"ssl-cipher", OPT_SSL_CIPHER, "SSL cipher to use (implies --ssl)",
     (gp_ptr*) &opt_ssl_cipher, NULL, 0,
     GET_STR, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
#endif /* HAVE_OPENSSL */
```

4) 在第3步里，头文件sslopt-longopts.h在定义与SSL有关的选项时使用了OPT\_SSL\_SSL、OPT\_SSL\_KEY等符号。这些符号的用途是充当相应的SSL选项的短选项代码，它们必须由程序来定义。具体到我们的sslclient.c文件，只要把下面这些代码添加到my\_opts数组定义的前面就行了：

```
#ifndef HAVE_OPENSSL
enum options
{
    OPT_SSL_SSL=256,
    OPT_SSL_KEY,
    OPT_SSL_CERT,
    OPT_SSL_CA,
    OPT_SSL_CAPATH,
    OPT_SSL_CIPHER
};
#endif
```

在编写你自己的程序时，如果某给定程序也为其他一些选项定义了短选项代码，请务必保证这些OPT\_SSL\_XXX形式的符号与其他的短选项代码有着不同的值。

5) 在第3步里，头文件sslopt-longopts.h定义了一些与SSL有关的选项，需要把这些选项的值保存在一些变量里才能在程序中使用。为了在程序里对这类变量进行声明，需要使用一条

#include指令把头文件sslopt-vars.h包括到程序里, 这条指令必须出现在my\_opts数组定义的前面。头文件sslopt-vars.h的内容如下所示:

```
#ifndef HAVE_OPENSSL
static my_bool opt_use_ssl = 0;
static char *opt_ssl_key = 0;
static char *opt_ssl_cert = 0;
static char *opt_ssl_ca = 0;
static char *opt_ssl_capath = 0;
static char *opt_ssl_cipher = 0;
#endif
```

6) 在get\_one\_option()例程里, 增加一行代码把头文件sslopt-case.h包括进来, 如下所示:

```
my_bool
get_one_option (int optid, const struct my_option *opt, char *argument)
{
    switch (optid)
    {
        case '?':
            my_print_help (my_opts);    /* print help message */
            exit (0);
        case 'p':                        /* password */
            if (!argument)               /* no value given, so solicit it later */
                ask_password = 1;
            else                          /* copy password, wipe out original */
            {
                opt_password = strdup (argument);
                if (opt_password == NULL)
                {
                    print_error (NULL, "could not allocate password buffer");
                    exit (1);
                }
                while (*argument)
                    *argument++ = 'x';
            }
            break;
    }
    #include <sslopt-case.h>
    return (0);
}
```

头文件sslopt-case.h的内容是一些用在switch()语句里的case子句。这些子句负责检测程序的输入参数向量里是否有SSL选项, 如果有, 就把opt\_use\_ssl变量设置为1。这个头文件的内容如下所示:

```
#ifndef HAVE_OPENSSL
case OPT_SSL_KEY:
case OPT_SSL_CERT:
```

```

case OPT_SSL_CA:
case OPT_SSL_CAPATH:
case OPT_SSL_CIPHER:
/*
    Enable use of SSL if we are using any ssl option
    One can disable SSL later by using --skip-ssl or --ssl=0
*/
    opt_use_ssl= 1;
    break;
#endif

```

这样做的效果是：在选项处理工作结束后，程序代码可以通过检测opt\_use\_ssl变量值的办法来判断用户是否想使用安全化连接。

在按照以上步骤完成修改之后，MySQL C客户程序开发库中负责处理命令选项的load\_defaults()和handle\_options()函数就能识别出SSL选项并自动设置好它们的值了。现在，还剩下最后一件事情：在程序sslclient里增加一些代码。如果在选项文件或者命令行上给出的选项表明用户想建立一个SSL连接，这些代码将在sslclient程序开始连接MySQL服务器之前把SSL选项信息传递给有关的MySQL C API函数。这个工作可以调用mysql\_ssl\_set()函数来完成，这个函数必须安排在调用mysql\_init()函数之后、调用mysql\_real\_connect()函数之前。如下所示：

```

/* initialize connection handler */
conn = mysql_init (NULL);
if (conn == NULL)
{
    print_error (NULL, "mysql_init() failed (probably out of memory)");
    exit (1);
}

#ifdef HAVE_OPENSSL
/* pass SSL information to client library */
if (opt_use_ssl)
    mysql_ssl_set (conn, opt_ssl_key, opt_ssl_cert, opt_ssl_ca,
                  opt_ssl_capath, opt_ssl_cipher);
#endif

/* connect to server */
if (mysql_real_connect (conn, opt_host_name, opt_user_name, opt_password,
                       opt_db_name, opt_port_num, opt_socket_name, opt_flags) == NULL)
{
    print_error (conn, "mysql_real_connect() failed");
    mysql_close (conn);
    exit (1);
}

```

注意：不必测试mysql\_ssl\_set()函数是否返回了一个错误——如果传递给这个函数的信息有问题，就会在调用mysql\_real\_connect()时看到一条相应的出错信息。

现在，编译并链接`sslclient.c`文件以生成`sslclient`程序，然后运行它。如果`mysql_real_connect()`调用成功，就可以发出查询命令了。如果在运行`sslclient`程序的时候使用了适当的SSL选项，与MySQL服务器之间的通信就将在一个加密连接上进行。如果想知道情况是否如此，可以发出一个下面这样的查询：

```
SHOW STATUS LIKE 'Ssl_cipher'
```

如果`Ssl_cipher`变量的值非空，就说明正使用着某种加密算法。（为了让大家省点事，`sampdb`发行版本里的`sslclient`程序会发出这个查询并报告其结果。）

## 6.8 嵌入式MySQL服务器程序开发库的使用

MySQL 4新增了一个名为`libmysqld`的嵌入式MySQL服务器程序开发库。这个库其实就是一个完备的MySQL服务器，但有关例程被编写成能够链接到各种应用程序里的形式。利用这个库，我们就可以开发出能够独立运行并完成各种MySQL数据库操作的应用程序来；而利用MySQL C客户程序开发库编写出来的应用程序只能作为客户程序通过网络连接到另外一个服务器程序才能进行各种数据库操作。

如果想编写一个内建有嵌入式MySQL服务器的应用程序，就必须满足两个条件。首先，机器上必须安装嵌入式MySQL服务器程序开发库：

- 如果是从源代码开始去安装MySQL软件的，请在运行`configure`脚本时使用`--with-embedded-server`选项激活这个库。
- 如果是从二进制发行版本开始去安装MySQL软件的，请选择一个Max发行版本（如果相应的非Max发行版本没有收录`libmysqld`的话）。
- 如果是用RPM文件来安装MySQL软件的，请务必把对应于嵌入式MySQL服务器的各个RPM文件都安装到机器里去。

其次，还需要在编写的应用程序里增加一些用来启动和关闭MySQL服务器的代码。

在做好这两项工作后，只需编译应用程序并把它与嵌入式MySQL服务器程序开发库（`-lmysqld`）（注意，不是普通的MySQL C客户程序开发库（`-lmysqlclient`））链接起来就行了。事实上，如果在编写应用程序的时候使用了来自`libmysqld`库的API例程，那么，既可以把这个应用程序与`libmysqld`库链接在一起，也可以把它与`mysqlclient`库链接在一起，你将分别得到一个嵌入版和一个客户/服务器版的应用程序——`libmysqld`库和`mysqlclient`库的开发者们早已预见到了这种情况并做出了必要的安排：`mysqlclient`库包含着与`libmysqld`库例程有着同样的调用序列的接口函数，只不过`mysqlclient`库里的这类接口函数没有具体的实现代码而已，它们只是一些什么也不做的“哑”例程。

### 6.8.1 编写一个内建有嵌入式MySQL服务器的应用程序

编写一个内建有嵌入式MySQL服务器的应用程序与编写一个将运行在客户/服务器环境里的客户程序并没有太大的区别。事实上，即使当初打算编写的是一个将运行在客户/服务器环境里的客户程序，也可以用嵌入式MySQL服务器程序开发库轻松地对它进行转换。下面，我们就以

把client4程序转换为嵌入式应用程序embapp为例来介绍具体的修改步骤。先用client4.c文件拷贝出一份embapp.c文件，然后按以下步骤修改embapp.c文件：

- 1) 在程序现有的MySQL头文件的基础上再增加一个mysql\_embed.h文件：

```
#include <my_global.h>
#include <mysql.h>
#include <mysql_embed.h>
#include <my_getopt.h>
```

2) 一个嵌入式应用程序同时包含着一个客户端和一个服务器端，所以它既可以处理供客户端使用的选项，也可以处理供服务器端使用的选项。比如说，假设这个名为embapp的嵌入式应用程序需要为它的客户部分去读取选项文件里的[client]和[embapp]选项组，就需要把client\_groups数组的定义修改为如下所示的样子：

```
static const char *client_groups[] =
{
    "client", "embapp", NULL
};
```

这些选项组里的选项将由load\_defaults()和handle\_options()函数按我们以前介绍过的方式进行处理。接下来，还要定义一组供服务器端使用的选项组。按照惯例，这组选项组将包括[server]、[embedded]和[appname\_SERVER]选项组，其中，appname是应用程序的名字。具体到embapp程序，供它专用的选项组就将是[embapp\_SERVER]，所以我们将把这组供服务器端使用的选项组定义为如下所示的样子：

```
static const char *server_groups[] =
{
    "server", "embedded", "embapp_SERVER", NULL
};
```

3) 在开始与服务器通信之前，先调用mysql\_server\_init()。最好在调用mysql\_init()函数之前做这件事。

4) 在结束与服务器的通信之后，再调用mysql\_server\_end()。最好在调用mysql\_close()函数之后做这件事。

完成上述修改后，源文件embapp.c里的main()函数将是如下所示的样子：

```
int
main (int argc, char *argv[])
{
    int opt_err;

    my_init ();
    load_defaults ("my", client_groups, &argc, &argv);

    if ((opt_err = handle_options (&argc, &argv, my_opts, get_one_option)))
        exit (opt_err);

    /* solicit password if necessary */
```



```
if (ask_password)
    opt_password = get_tty_password (NULL);

/* get database name if present on command line */
if (argc > 0)
{
    opt_db_name = argv[0];
    --argc; ++argv;
}

/* initialize embedded server */
mysql_server_init (0, NULL, (char **) server_groups);

/* initialize connection handler */
conn = mysql_init (NULL);
if (conn == NULL)
{
    print_error (NULL, "mysql_init() failed (probably out of memory)");
    exit (1);
}

/* connect to server */
if (mysql_real_connect (conn, opt_host_name, opt_user_name, opt_password,
    opt_db_name, opt_port_num, opt_socket_name, opt_flags) == NULL)
{
    print_error (conn, "mysql_real_connect() failed");
    mysql_close (conn);
    exit (1);
}

while (1)
{
    char    buf[10000];

    fprintf (stderr, "query> ");
    if (fgets (buf, sizeof (buf), stdin) == NULL)
        break;
    if (strcmp (buf, "quit\n") == 0 || strcmp (buf, "\\q\n") == 0)
        break;
    process_query (conn, buf);

    /* print prompt */
    /* read query */
    /* execute query */

/* disconnect from server */
mysql_close (conn);
/* shut down embedded server */
mysql_server_end ();
exit (0);
}
```

### 6.8.2 生成一个内建有嵌入式MySQL服务器的应用程序可执行二进制文件

如果想得到一个内建有嵌入式MySQL服务器的embapp可执行二进制文件，就需要用-lmysqld库而不是-lmysqlclient库来链接之。这里是mysql\_config工具大显身手的地方。就像它能告诉你需要使用哪些标志来链接MySQL C客户程序开发库那样，它也可以告诉你需要使用哪些标志来链接嵌入式MySQL服务器程序开发库：

```
% mysql_config --libmysqld-libs
-L'/usr/local/mysql/lib/mysql' -lmysqld -lz -lm
```

也就是说，用来生成嵌入版embapp程序的编译和链接命令将是：

```
% gcc -c 'mysql_config --cflags' embapp.c
% gcc -o embapp embapp.o 'mysql_config --libmysqld-libs'
```

到了这一步，就得到了一个嵌入式的应用程序，其中有需要用来访问MySQL数据库的一切东西。不过，还有几件事必须注意：当运行embapp程序的时候，千万不要让它与正在同一台机器里运行的其他MySQL服务器（客户/服务器模式下的也好，独立运行模式下的也好）使用相同的数据目录，以免导致系统崩溃；在UNIX系统上，必须通过一个在数据目录上有访问权限的账户去运行这个应用程序。如果是数据目录的属主身份登录的，想运行embapp程序当然不成问题。另一种做法是把embapp程序转换为一个setuid程序，让它在启动时把自己的用户ID自动切换为指定用户。比如说，如果想让embapp程序在运行时具备用户mysqladm的权限，请以根用户root的身份执行以下命令：

```
# chown mysqladm embapp
# chmod 4755 embapp
```

如果后来又想生成一个将在客户/服务器上下文里运行的非嵌入版embapp程序，只要把它与MySQL C客户程序开发库链接起来就行了。有关的编译和链接命令如下所示：

```
% gcc -c 'mysql_config --cflags' embapp.c
% gcc -o embapp embapp.o 'mysql_config --libs'
```

MySQL C客户程序开发库包含有mysql\_server\_init()和mysql\_server\_end()函数的“哑”版本，所以不会发生链接错误。

## 6.9 其他论题

在这一节里，我们将对几个查询处理工作有关的问题进行论述，它们是一些不太适合放到本章前面各节里的论题：

- 如何在已经根据结果集元数据确定结果集数据适合进行某种计算的情况下对它们进行计算。
- 如何对将被插入到查询命令里去的数据值中的特殊字符进行编码。
- 如何对二进制数据进行处理。
- 如何获取关于数据表结构的信息。
- MySQL程序设计工作中的常见错误及预防办法。

### 6.9.1 在结果集上进行计算

到目前为止，我们只介绍了结果集元数据在打印数据行方面的应用，但对结果集的使用肯定不会总是打印它这么简单。比如说，可能需要对结果集里的数据进行统计学分析，或者需要使用结果集元数据来判断结果集数据是否符合某种预期，等等。那么，这里所说的“某种预期”到底是什么样的呢？一种比较常见的情况是打算对某些数据列进行数值计算，因而需要验证它们确实包含着数值。

下面是一个简单的summary\_stats()函数的源代码，这个函数将根据给定的结果集和数据列下标对该数据列里的数据进行一些统计学分析并把其结果报告给你。这个函数还会告诉你那个数据列里缺失了多少个数据，即里面有多少个NULL值。这些计算要求有关数据满足两个条件，而summary\_stats()函数将利用结果集元数据来完成必要的验证工作：

- 给定数据列必须是存在的。也就是说，所给出的数据列下标值必须落在结果集里的数据列个数范围内。这个范围从0到mysql\_num\_fields()-1。
- 给定数据列必须包含着数值。

如果这两个条件得不到满足，summary\_stats()函数将简单地打印一条出错信息并返回。下面就是这个函数的实现代码：

```
void
summary_stats (MYSQL_RES *res_set, unsigned int col_num)
{
    MYSQL_FIELD      *field;
    MYSQL_ROW        row;
    unsigned int      n, missing;
    double            val, sum, sum_squares, var;

    /* verify data requirements: column must be in range and numeric */
    if (col_num < 0 || col_num >= mysql_num_fields (res_set))
    {
        print_error (NULL, "illegal column number");
        return;
    }
    mysql_field_seek (res_set, col_num);
    field = mysql_fetch_field (res_set);
    if (!IS_NUM (field->type))
    {
        print_error (NULL, "column is not numeric");
        return;
    }

    /* calculate summary statistics */

    n = 0;
    missing = 0;
    sum = 0;
```

```

sum_squares = 0;

mysql_data_seek (res_set, 0);
while ((row = mysql_fetch_row (res_set)) != NULL)
{
    if (row[col_num] == NULL)
        missing++;
    else
    {
        n++;
        val = atof (row[col_num]); /* convert string to number */
        sum += val;
        sum_squares += val * val;
    }
}
if (n == 0)
    printf ("No observations\n");
else
{
    printf ("Number of observations: %lu\n", n);
    printf ("Missing observations: %lu\n", missing);
    printf ("Sum: %g\n", sum);
    printf ("Mean: %g\n", sum / n);
    printf ("Sum of squares: %g\n", sum_squares);
    var = ((n * sum_squares) - (sum * sum)) / (n * (n - 1));
    printf ("Variance: %g\n", var);
    printf ("Standard deviation: %g\n", sqrt (var));
}
}

```

请注意那个出现在进入mysql\_fetch\_row()循环之前的mysql\_data\_seek()调用，它的作用是定位到结果集里的第一个数据行处，这是为了应付你可能需要用summary\_stats()函数对同一个结果集做多次处理（比如连续地对多个数据列进行统计分析）的情况而准备的。这样，summary\_stats()函数的每一次执行都将从结果集中的第一个数据行开始进行统计分析。mysql\_data\_seek()函数要求结果集必须是通过mysql\_store\_result()调用而创建出来的。如果结果集是通过mysql\_use\_result()调用而创建出来的，就只能按顺序对各数据行进行处理，并且只能处理它们一次。

虽说summary\_stats()函数比较简单，但可以利用它的思路来进行更复杂的计算，比如两个数据列上的最小方差回归或者标准统计（例如t测试或差异分析）等。

### 6.9.2 对查询命令中的特殊字符进行编码

如果不加处理地把包含引号、NULL字节、反斜线等字符的数据值放到查询命令字符串里，则在执行这个查询的时候就会遇到麻烦。下面将对这一问题的本质和解决办法进行介绍。

假设在构造的SELECT查询里用到了一个变量name\_val，而这个变量的值是一个以NULL字

节结尾的字符串，如下所示：

```
char query[1024];

sprintf (query, "SELECT * FROM mytbl WHERE name='%s'", name_val);
```

那么，万一变量name\_val的值是“O'Malley, Brian”之类的东西，所构造出来的查询命令字符串就将是非法的，因为一个引号字符出现在了用引号引起来的字符串里：

```
SELECT * FROM mytbl WHERE name='O'Malley, Brian'
```

因此，必须把字符串里面的引号当做特殊情况来处理，否则，服务器就会把它解释为字符串的结束标志。ANSI SQL给出的解决办法是双写字符串中的引号字符。MySQL既支持这一做法，也允许用一个反斜线字符对引号字符进行转义，所以上面查询的以下两种写法在MySQL里都是正确的：

```
SELECT * FROM mytbl WHERE name='O''Malley, Brian'
SELECT * FROM mytbl WHERE name='O\'Malley, Brian'
```

在查询命令里使用二进制数据（例如，当应用程序试图把图像存入数据库时）也可能引起类似的问题。因为二进制值允许包含任何字符（引号和反斜线字符也不例外），所以把它们直接放到查询命令里并不安全。

这些问题可以使用mysql\_real\_escape\_string()函数解决，这个函数将对特殊字符进行编码，使它们适合用在引号引起来的字符串里。mysql\_real\_escape\_string()函数将把NULL字节、单引号（'）、双引号（"）、反斜线（\）、换行符、回车符、Ctrl-Z（Ctrl-Z是Windows里的特殊字符，它通常被用做文件结束标志）等字符视为特殊字符。

那么，哪些场合应该使用mysql\_real\_escape\_string()函数呢？最保险的答案是“任何场合”。不过，如果你对数据的格式和内容都很有把握的话（比如，事先做过其他检查），就不必对它们进行编码。比如说，如果所处理的字符串代表的是一些合法的电话号码——即仅由数字和连字符组成，当然没必要去调用这个函数。至于其他情况，还是慎重些为好。

mysql\_real\_escape\_string()函数将把一个特殊字符编码为一个由两个字符组成的序列，序列中的第一个字符是反斜线（\）。比如说，NULL字节将被编码为“\0”，其中的“0”是可打印的ASCII字符“0”而不再是NULL字节。反斜线、单引号、双引号将分别被编码为“\\”、“\'”和“\"”。

mysql\_real\_escape\_string()函数的调用方法如下所示：

```
to_len = mysql_real_escape_string (conn, to_str, from_str, from_len);
```

mysql\_real\_escape\_string()对from\_str进行编码并把结果写到to\_str里。它还会在to\_str的末尾加上一个NULL字节，这为我们用strcpy()、strlen()、printf()等函数来进一步处理结果字符串提供了方便。

from\_str指向一个char缓冲区，其内容是被编码的字符串，这个字符串允许包含包括二进制数据在内的任何东西。to\_str指向一个已经存在的char缓冲区，其内容将是编码后的结果字符串。注意：千万不要把一个未经初始化的指针或者一个NULL指针传递给mysql\_real\_escape\_string()函数的to\_str参数，这个函数是不会替你去分配内存的！to\_str缓冲区的长度至少要有(form\_len \*



2)+1个字节。(在最坏的情况下,源字符串from\_str中的每一个字符都需要被编码为一个由两个字节组成的序列,多出来的那个字节是给字符串结束标志NULL字节保留的。)

from\_len和to\_len都是unsigned long类型的值。from\_len给出的是from\_str缓冲区里的数据长度,这个长度值是必不可少的(因为from\_str允许包含NULL字节,所以不能把它当做以NULL字节结尾的字符串来对待)。to\_len,也就是mysql\_real\_escape\_string()函数的返回值,是经过编码的结果字符串的真实长度,但作为其结束标志的NULL字节不计算在内。

当mysql\_real\_escape\_string()调用返回时,因为from\_str里的NULL字节全都被编码为可打印的“\0”序列,所以我们可以把to\_str里的编码结果当做一个以NULL字节结尾的字符串来使用。

现在,我们来改写刚才的SELECT查询构造代码,使它在遇到像“O'Malley, Brain”这样包含着引号的值时也能够正确工作。可以像下面这样做:

```
char query[1024], *p;

p = strcpy (query, "SELECT * FROM mytbl WHERE name='");
p += strlen (p);
p += mysql_real_escape_string (conn, p, name, strlen (name));
*p++ = '\'';
*p = '\0';
```

必须承认,上面这段代码有些难看难懂。下面这段代码就好多了,但这是有代价的——得多使用一个缓冲区:

```
char query[1024], buf[1024];

(void) mysql_real_escape_string (conn, buf, name, strlen (name));
sprintf (query, "SELECT * FROM mytbl WHERE name='%s'", buf);
```

mysql\_real\_escape\_string()函数最早出现于MySQL 3.23.14版本。在此之前,对特殊字符进行编码的工作可以用mysql\_escape\_string()函数来完成,如下所示:

```
to_len = mysql_escape_string (to_str, from_str, from_len);
```

这两个函数之间的区别是:mysql\_real\_escape\_string()在进行编码时使用的是当前连接上的字符集,而mysql\_escape\_string()使用的则是默认字符集(所以它没有连接句柄参数)。如果想让源代码在MySQL软件的任何版本下都能编译,请把下面这个代码段包括到源文件里去:

```
#if !defined(MYSQL_VERSION_ID) || (MYSQL_VERSION_ID<32314)
#define mysql_real_escape_string(conn,to_str,from_str,len) \
    mysql_escape_string(to_str,from_str,len)
#endif
```

然后,在编程时总使用mysql\_real\_escape\_string(),如果这个函数不存在,上面的#define指令将会把它映射为mysql\_escape\_string()。

### 6.9.3 对图像数据进行处理

有项工作是必须要靠mysql\_real\_escape\_string()函数才能完成的,那就是把图像数据加载到数

据表里去, 本小节将介绍具体的做法。(这里的讨论也同样适用于任何其他形式的二进制数据。)

假设现在要把一些从文件里读出来的图像存入一个名为picture的数据表, 每个图像都有一个独一无二的标识符。二进制数据适合用BLOB类型的数据列来保存, 所以使用一个如下所示的数据表:

```
CREATE TABLE picture
(
    pict_id      INT NOT NULL PRIMARY KEY,
    pict_data    BLOB
);
```

把图像加载到picture数据表里的工作由load\_image()函数完成, 它将把已打开的给定文件里的图像数据和给定的标识号一起插入到数据表里。下面就是它的代码:

```
int
load_image (MYSQL *conn, int id, FILE *f)
{
    char          query[1024*100], buf[1024*10], *p;
    unsigned long  from_len;
    int            status;

    sprintf (query,
            "INSERT INTO picture (pict_id,pict_data) VALUES (%d,'",
            id);
    p = query + strlen (query);
    while ((from_len = fread (buf, 1, sizeof (buf), f)) > 0)
    {
        /* don't overrun end of query buffer! */
        if (p + (2*from_len) + 3 > query + sizeof (query))
        {
            print_error (NULL, "image too big");
            return (1);
        }
        p += mysql_real_escape_string (conn, p, buf, from_len);
    }
    *p++ = '\\';
    *p++ = ')';
    status = mysql_real_query (conn, query, (unsigned long) (p - query));
    return (status);
}
```

从图像处理的角度讲, load\_image()函数分配的查询缓冲区并不大(100KB), 所以它只能用来加载比较小的图像。在实际工作中, 应该根据图像文件的尺寸动态地为这个缓冲区分配内存。

从数据库取出图像数据(或者其他二进制数据)不像当初把它们放进去那么难: 构成图像的数据值全都在MYSQL\_ROW变量里, 其长度可以通过调用mysql\_fetch\_lengths()函数的办法获得。只是一定要按计数字符串来对待这些数据值, 千万不要把它们当成以NULL字节结尾的字符串。

#### 6.9.4 获取关于数据表结构的信息

在MySQL里，可以使用以下查询中的任何一个去获取关于数据表结构的信息，它们是等价的：

```
SHOW COLUMNS FROM tbl_name;
SHOW FIELDS FROM tbl_name;
DESCRIBE tbl_name;
EXPLAIN tbl_name;
```

类似于SELECT语句，这些语句都将返回一个结果集。要想知道数据表的构造情况，只需对结果集里的数据行进行处理，把所需要的信息从中提取出来就行了。比如说，当在mysql客户程序里发出一个DESCRIBE president语句的时候，它将返回如下所示的信息：

```
mysql> DESCRIBE president;
+-----+-----+-----+-----+-----+-----+
| Field      | Type          | Null | Key | Default      | Extra |
+-----+-----+-----+-----+-----+-----+
| last_name  | varchar(15)   |      |     |               |       |
| first_name | varchar(15)   |      |     |               |       |
| suffix     | varchar(5)    | YES  |     | NULL          |       |
| city       | varchar(20)   |      |     |               |       |
| state      | char(2)       |      |     |               |       |
| birth      | date          |      |     | 0000-00-00    |       |
| death     | date          | YES  |     | NULL          |       |
+-----+-----+-----+-----+-----+-----+
```

当在自行开发的客户程序里发出同样的查询时，也将获得同样的信息（没有表格框线）。如果只想了解某特定数据列的情况，把它的名字添上就行了，如下所示：

```
mysql> DESCRIBE president birth;
+-----+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default      | Extra |
+-----+-----+-----+-----+-----+-----+
| birth | date |      |     | 0000-00-00    |       |
+-----+-----+-----+-----+-----+-----+
```

#### 6.9.5 MySQL程序设计工作中的常见错误及预防办法

本小节将对MySQL C API程序设计工作中的一些常见错误及预防办法进行讨论。（这些问题都取材于MySQL邮件列表，而且具有相当的普遍性。）

##### 1. 错误1——使用未经初始化的连接句柄指针

在前面那些程序示例里，我们一直是用一个NULL参数来调用mysql\_init()函数的。NULL参数将使mysql\_init()为一个MYSQL结构分配内存并对之进行初始化，最后返回一个指向该结构的指针。mysql\_init()函数的另一种调用办法是把一个指向已经存在着的某个MYSQL结构的指针传递给它。此时，mysql\_init()将对给定MYSQL结构进行初始化并返回一个指向它的指针，省略了内存分配步骤。如果想使用第二种办法，就一定要注意下面讨论的问题。

当把一个指针传递给mysql\_init()的时候，它必须真的指向某个东西。请看下面这段代码：

```
main ()
{
    MYSQL      *conn;

    mysql_init (conn);
    ...
}
```

看出问题了吗？mysql\_init()收到了一个指针，可这个指针到底指向什么地方却不清楚。conn是一个局部变量，操作系统不会对它进行初始化（清零）。因此，当main()开始执行的时候，它可能指向任何地方，而mysql\_init()则会把一个MYSQL结构的初始信息写到这个指针当时指向的地方。如果运气好，conn将指向这个程序的地址空间以外的某个地方，操作系统将立刻发现这一问题并中止程序的执行，你也就早早地知道了自己的代码有问题。如果运气没这么好，conn将指向程序稍后才会用到的某些数据，而你在这个程序真的用到那些数据之前是不会发现有什么不对劲的。在这种情况下，程序执行出错的“现场”与错误真正发生的位置往往相距甚远，因而给调试工作带来极大的困难。

下面这段代码也犯了同样的错误：

```
MYSQL      *conn;

main ()
{
    mysql_init (conn);
    mysql_real_connect (conn, ...)
    mysql_query(conn, "SHOW DATABASES");
    ...
}
```

这段代码里的conn是一个全局变量，所以操作系统会在这个程序开始执行之前把它初始化为0，即NULL。于是，mysql\_init()将看到一个NULL参数，它将为一个MYSQL结构分配内存、对它进行初始化、返回一个新的连接句柄。但是，因为没有对conn变量进行赋值，所以它的值仍将是NULL。这样一来，当把conn传递给一个只有在连接句柄参数不是NULL值的前提下才能正确执行的MySQL C API函数时，程序就崩溃了。上面这两段代码中的错误有着同样的本质，只要能保证不使用未经初始化的连接句柄指针，它们就不会发生。比如说，可以像下面这样把conn初始化为指向某个肯定存在的MYSQL结构的地址：

```
MYSQL conn_struct, *conn = &conn_struct;
...
mysql_init (conn);
```

不过，更稳妥（也更简单！）的做法是明确地给mysql\_init()函数传递一个NULL参数，让它替你去分配MYSQL结构，然后再把它的返回值赋值给conn。如下所示：

```
MYSQL *conn;
...
conn = mysql_init (NULL);
```

无论采用哪种做法，千万不要忘记检查mysql\_init()的返回值是不是NULL（参见“错误2”）。

## 2. 错误2——没有对函数调用的返回值进行检查

只要某个函数调用有失败的可能性，就一定要对它的返回状态进行检查。下面这段代码没有这样做：

```
MYSQL_RES *res_set;
MYSQL_ROW row;

res_set = mysql_store_result (conn);
while ((row = mysql_fetch_row (res_set)) != NULL)
{
    /* process row */
}
```

看出问题了吗？万一mysql\_store\_result()调用失败，res\_set就将是NULL，此时，那个while循环根本不应该进入——把NULL传递给mysql\_fetch\_row()往往会导致程序崩溃。因此，对于会返回结果集的那些API函数，一定要对它们的返回值进行检查，以保证自己真的有东西可以处理。

这一原则适用于一切有可能调用失败的函数。如果跟在某个函数后面的代码只有在这个函数调用成功的前提下才能正确执行，就必须对它的返回值进行检查，并安排适当的出错处理代码。如果想当然地认为它肯定会调用成功，那么问题将迟早会发生。

## 3. 错误3——没有考虑到NULL数据列值的问题

千万不要忘记检查mysql\_fetch\_row()调用所返回的MYSQL\_ROW数组中的数据列值是不是NULL指针。比如说，在某些机器上，一旦row[i]是NULL，下面这段代码就会崩溃：

```
for (i = 0; i < mysql_num_fields (res_set); i++)
{
    if (i > 0)
        fputc ('\t', stdout);
    printf ("%s", row[i]);
}
fputc ('\n', stdout);
```

这个错误最让人头疼的地方是，printf()函数的某些版本会把NULL指针打印为“(null)”而不是报告出错或者崩溃，所以很多人都不把这个问题当回事。如果你把自己编写的程序送给了一位朋友，可这个程序却因为这位朋友的printf()函数不那么“宽容”而在他的机器里崩溃了的话，你的朋友会怎么看你？上面的循环应该写成下面这个样子：

```
for (i = 0; i < mysql_num_fields (res_set); i++)
{
    if (i > 0)
        fputc ('\t', stdout);
    printf ("%s", row[i] != NULL ? row[i] : "NULL");
}
fputc ('\n', stdout);
```

有没有不需要检查数据列值是否为NULL的场合呢？有，但是只有一个：如果已经根据对应



于某数据列的MYSQL\_FIELD结构确定该数据列上的IS\_NOT\_NULL()为真，即该数据列不包含NULL值，当然不必再对其中的数据值进行检查了。

#### 4. 错误4——供函数调用存放结果的缓冲区不合要求

有些MySQL C API函数在执行时需要用到由你提供的缓冲区，这类函数大都要求所提供的缓冲区必须真正存在。下面这段代码就违反了这一原则：

```
char *from_str = "some string";  
char *to_str;  
unsigned long len;  
  
len = mysql_real_escape_string (conn, to_str, from_str, strlen (from_str));
```

看出问题了吗？to\_str本该指向一个已经存在着的缓冲区，但它却没有；事实上，它还没有被初始化，我们根本无法知道它指向哪里。如果不想让mysql\_real\_escape\_string()把编码结果随机地写在内存中的某个地方，就千万不要把未经初始化的指针传递给它的to\_str参数。

## 第7章 MySQL应用程序设计接口：Perl DBI

在这一章里，我们将重点介绍MySQL的Perl语言程序设计接口DBI的使用方法。如果了解DBI的工作原理和体系结构（尤其是它与MySQL的C API和PHP API的对比），请参见第5章中的讨论。

在本书的第1章里，我们创建了一个样板数据库sampdb，并在其中为考试记分项目和美国历史研究会分别创建了一些数据表。在这一章里，就将以这个样板数据库中的数据表为例来展开讨论。为了更好地完成这一章的学习，我们希望大家对Perl有一定的了解。虽说不熟悉Perl也完全能参照本章中的示例代码编写出一些脚本程序来，但一本好的Perl教科书却肯定是一项非常划算的投资。由Wall、Christiansen和Orwant撰写的*Programming Perl, Third Edition*（Perl语言程序设计，第3版）就是一本这样的好书（O'Reilly，2000）。

DBI的最新版本是1.32，但本章中的大部分讨论也同样适用于较早的1.xx系列版本。在讨论过程中，如果涉及到一些早期版本所不具备的功能，我们会把它们注明出来。DBI要求运行在Perl 5.005\_03或更高的版本上（1.21版之前的DBI要求运行在Perl 5.004\_05或更高的版本上）。系统中还必须安装有配合Perl脚本使用的DBD::mysql模块。如果还想利用本章介绍的技巧编写一些基于Web的DBI脚本，就必须把CGI.pm模块也安装好。在这一章里，CGI.pm模块将与Web服务器Apache配合使用。如果你现在还没有安装好上面提到的这几个软件，请参阅附录A。获得本章各示例程序代码的办法也在可以在附录A里查到——它们是sampdb发行版本的组成部分之一，如果下载了sampdb发行版本，就用不着自己动手输入示例脚本的代码了。与本章有关的脚本都放在sampdb发行版本的perlapi目录里。

限于篇幅，将只对本章内容涉及到的那些Perl DBI方法和变量做简单的解释，还有很多方法和变量无法一一介绍。附录G对DBI模块中的每一种方法和变量都做了详细的介绍。如果在今后的程序设计工作中需要用到其他的DBI方法或变量，可以把本书的附录G当做使用手册来参考。在线文档可以用下面的命令查阅到：

```
% perldoc DBI
% perldoc DBI::FAQ
% perldoc DBD::mysql
```

在数据库驱动程序（database driver，DBD）层次上，MySQL驱动程序是在MySQL C客户程序开发库的基础上编写出来的，所以它们有很多相似的地方。对MySQL C客户程序开发库的详细介绍见本书的第6章。

### 7.1 Perl语言脚本程序的特点

Perl脚本都是普通的文本文件，可以用任何一种文本编辑器来进行编辑。这一章里的Perl脚本都遵守UNIX系统上的脚本编写规定：在第一行用记号“#!”给出一个路径名，路径名上的程

序将被用来执行这个脚本。我在脚本的第一行是这样写的:

```
#!/usr/bin/perl -w
```

如果你使用的也是UNIX系统,请把Perl在系统上的路径名(可能是/usr/local/bin/perl5或/opt/bin/perl)写在所编写的Perl脚本的第一行。如果路径有误,Perl脚本将无法在系统上正确执行。

注意,在“#!”后面还多加了一个空格。这是因为有些系统只把4字节序列“#!/”当做一个有魔力的记号——如果“#!”和“/”之间没有空格,这个脚本就会被认为是一个shell脚本而不是一个Perl脚本。

-w选项告诉Perl在发现你使用了有问题的语言结构或者执行了有疑问的操作(比如试图打印没有初始化过的变量)时发出一个警告。这种提醒有助于减少脚本程序中的错误。

可以像下面这样来执行一个Perl脚本,这个办法适用于允许使用Perl脚本的任何系统:

```
% perl -w myscript.pl
```

在UNIX系统上,有一种办法能让你用不着在命令行上写出perl程序就能执行Perl脚本:可以用chmod命令把脚本设置为可执行模式,如下所示:

```
% chmod +x myscript.pl
```

以后,只需给出这个脚本的名字就能执行它了,如下所示:

```
% ./myscript.pl
```

在后面的内容里,将统一采用这种方式来执行Perl脚本。如果脚本文件保存在当前目录(以符号“.”代表)里而shell却没有把这个当前目录添加到其搜索路径里去,脚本文件名的前面就必须有“./”记号。否则,脚本文件名前面的“./”记号就可以省略:

```
% myscript.pl
```

在基于Windows NT的系统(NT、2000和XP)上,人们经常采用在Perl和以.pl结尾的文件名之间建立一个文件名关联的办法来简化Perl脚本的执行操作。以ActiveState Perl为例,它的安装程序允许你建立一个文件名关联,让Perl去负责执行所有以.pl结尾的文件。在做了这样的设置之后,只需在命令行上给出Perl脚本的名字就能执行它们了:

```
C:\> myscript.pl
```

## 7.2 Perl DBI概述

本节将介绍DBI的背景知识——这些知识是你在编写自己的脚本或者阅读别人写的脚本时必须掌握的。如果你已经对DBI比较熟悉,可以直接跳到第7.3节。

### 7.2.1 DBI数据类型

Perl DBI API与第6章介绍的C语言客户程序开发库有很多相似相通的地方。在使用C语言客户程序开发库的时候,需要调用各种函数,与MySQL有关的数据大都需要通过指向结构或数组

的各种指针来访问。在使用DBI API的时候，还得用到各种函数和指向各种结构的指针，只是它们的称呼都发生了变化：函数现在被称为“方法”（method）、指针被称为“引用”（reference）、指针变量被称为“句柄”（handle），句柄指向的结构被称为“对象”（object）。

DBI句柄可以细分为很多种，它们在DBI文档里大都有一个约定俗成的称呼，表7-1列出了其中最常见的几种。这些句柄的用法将随着我们学习的深入而越来越清晰。有些常用的非句柄变量也有约定俗成的称呼（见表7-2）。这一章其实用不到这么多的变量，但当你阅读别人写的DBI脚本时，知道这些变量的含义就有用了。

表7-1 Perl DBI常用的句柄变量

名 称	含 义
\$dbh	一个指向某数据库对象的句柄
\$sth	一个指向某语句（查询）对象的句柄
\$fh	一个指向某打开文件的句柄
\$h	一个“普通”的句柄，它的含义要由上下文来决定

表7-2 Perl DBI常用的非句柄变量

名 称	含 义
\$rc	逻辑操作的返回代码
\$rv	整数操作的返回值
\$rows	数据行的个数
@ary	一个数组（列表），它代表着由查询命令返回的一个数据行

### 7.2.2 一个简单的DBI脚本

下面从一个简单的脚本dump\_members.pl开始，这个脚本演示了DBI程序设计中的几个标准概念，比如与MySQL服务器的连接和断开、发出数据库查询命令、对数据进行检索处理等。这个脚本将以制表符分隔格式输出一份“美国历史研究会”的会员名单。可我们现在关心的并不是它的输出格式是否美观，眼下最重要的是把DBI脚本的框架弄明白。

下面是dump\_members.pl脚本的源代码：

```
#!/usr/bin/perl -w
# dump_members.pl - dump Historical League's membership list

use strict;
use DBI;

my $dsn = "DBI:mysql:sampdb:cobra.snake.net"; # data source name
my $user_name = "sampadm"; # user name
my $password = "secret"; # password

# connect to database
my $dbh = DBI->connect ($dsn, $user_name, $password,
```

```

{ RaiseError => 1, PrintError => 0 });

# issue query
my $sth = $dbh->prepare ("SELECT last_name, first_name, suffix, email,"
    . "street, city, state, zip, phone FROM member ORDER BY last_name");
$sth->execute ();

# read results of query, then clean up
while (my @ary = $sth->fetchrow_array ())
{
    print join ("\t", @ary), "\n";
}
$sth->finish ();

$dbh->disconnect ();
exit (0);

```

如果你想试用一下这个脚本，可以使用sampdb发行版本里的拷贝，也可以用一个文本编辑器来亲手创建它。（如果使用的是文字处理程序，请记得把脚本保存为普通文本格式，而不要把它们保存为文字处理程序的默认格式——很可能是只有这种文字处理程序能看懂的二进制格式。）另外，可能还要对脚本代码里的连接参数（比如主机名、数据库名、用户名、口令等）做些修改。（只要是用到了连接参数的脚本，都得对那些参数做相应的修改。）在第7.2.9节，我们将把连接参数放入一个选项文件，这样就用不着再把连接参数硬编码到脚本程序里了。

下面，我们对这个脚本做逐行的分析。它在第一行给出了一个标准的Perl路径：

```
#!/usr/bin/perl -w
```

这一行是本章中的每一个脚本都有的内容，所以以后介绍示例脚本的时候将不再重复说明了。

在脚本程序的开头对它的功能做一下介绍是一种良好的编程习惯，它能帮助别人了解脚本的用途，所以接下来的一行就是一条这样的注释：

```
# dump_members.pl - dump Historical League's membership list
```

在Perl脚本里，从“#”字符到行尾之间的内容将被认为是一条注释。注释允许出现在脚本程序的任意位置，用注释对脚本程序的各种过程做出解释是一种良好的编程习惯。

接下来是两条use语句：

```
use strict;
use DBI;
```

use strict将使Perl运行在严格模式，即要求使用变量之前必须先对它们做出声明。在编写脚本程序的时候，可以不使用这条use strict命令，但写上它有助于捕获错误，所以建议大家在自己的脚本里永远要加上这一行。比如说，如果在脚本程序的前面声明了一个变量\$my\_var，但在后面的代码里把它错写为\$mv\_var，运行在严格模式下的脚本就会在执行时报告出一条如下所示的出错信息：



Global symbol "\$mv\_var" requires explicit package name at line n

当看到这条出错信息的时候，你就会想：“哦，怎么回事，这个脚本里不应该有\$mv\_var变量呀！”于是，你会去检查脚本程序的第n行，并发现是自己把\$my\_var错写为\$mv\_var了。这等于给了你一个改正错误的机会。如果在脚本程序的开头没有写上use strict语句，Perl就不会认为变量\$mv\_var是一个错误，它将默默地创建出这个名为\$mv\_var的变量并把它赋值为undef（意思是“未定义”），而让你自己去疑惑为什么这个脚本没有达到预定的效果。

use DBI将使Perl加载上必要的DBI模块。要是缺少了这一行，DBI脚本就将无法正常工作。用不着细致地指定将要使用哪一个DBD级的数据库驱动程序模块，因为DBI会在你尝试连接MySQL服务的时候自动判定并激活正确的数据库驱动程序模块。

因为这个脚本将运行在严格模式下，所以我们必须用关键字“my”对将要用到的每一个变量做出声明（可以把它想像成脚本程序在说“我明确地把这些东西声明为我的变量”）。接下来的脚本代码先对连接参数变量进行了声明和赋值，然后再用它们去连接数据库：

```
my $dsn = "DBI:mysql:sampdb:cobra.snake.net";    # data source name
my $user_name = "sampadm";                      # user name
my $password = "secret";                        # password

# connect to database
my $dbh = DBI->connect ($dsn, $user_name, $password,
                        { RaiseError => 1, PrintError => 0 });
```

connect()是DBI类的一个方法，所以要用DBI->connect()形式来调用。你用不着追究它到底代表着什么意思，它只是面向对象程序设计语言用来调用类方法的一种手段而已。（如果你真想知道的话，它表明connect()是一个“属于”DBI的函数。）需要向connect()提供以下几个输入参数：

- **数据源 (data source)**。也叫做数据源名 (data source name, DSN)。DSN的格式取决于具体使用的DBD模块。就MySQL数据库的驱动程序模块而言，允许使用的DSN格式有以下几种：

```
DBI:mysql:db_name
DBI:mysql:db_name:host_name
```

DBI的大小写无关紧要，但mysql却必须是小写。*db\_name*是准备使用的数据库的名字，*host\_name*是MySQL服务器在其上运行的主机名。如果没有给出主机名，它的默认值将是localhost。（事实上，还有另外几种数据源格式可供选用，我们将在第7.2.9节对它们进行介绍。）

- **MySQL账户的用户名和口令**。这两个参数就用不着多做解释了。
- **一个用来设定其他连接属性的可选参数**。如果给出了这个参数，它将决定DBI的出错处理行为。这个参数必须是一个指向一个散列列表（即元素为键/值对的列表）的引用指针，散列列表将给出这个连接的其他属性（如果有的话）的名称和值。我们给出的那个奇怪的构造创建了一个这样的散列列表引用指针：它激活了RaiseError属性，禁用了PrintError属性。这种设置将使DBI去检查和处理与数据库有关的执行出错，如果它真的检测到一个出错，

就会在显示一条出错信息后退出执行。(这正是为什么在这个dump\_members.pl脚本里看不到出错处理代码——DBI将替你去处理它们。)我们将在第7.2.3节介绍几种其他的出错处理方法。

如果connect()调用成功,它将返回一个数据库句柄,我们再把这个句柄赋值给变量\$dbh。在默认情况下,如果调用失败,connect()将返回undef。但因为这个脚本激活了RaiseError属性,所以如果真的在connect()的调用过程中出现了错误,DBI就会在显示一条出错信息后退出执行。(其他DBI方法也是如此,稍后将介绍它们在执行出错时会返回一个什么样的值。但如果激活了RaiseError属性,它们将从方法调用中直接退出执行而根本不会返回到脚本。)

在连接上数据库之后,dump\_members.pl脚本将发出一条SELECT查询去检索“美国历史研究会”的会员名单,然后再执行一个循环语句以处理它检索到的每一个数据行。这些数据行构成了这个SELECT查询的结果集(result set,也叫“结果集合”)。为了进行SELECT查询,得先把有关的查询命令准备好,然后再执行它:

```
# issue query
my $sth = $dbh->prepare ("SELECT last_name, first_name, suffix, email,"
    . "street, city, state, zip, phone FROM member ORDER BY last_name");
$sth->execute ();
```

使用数据库句柄\$dbh调用了prepare()方法,它将把SQL语句传递给相关的数据库驱动程序去进行执行前的预处理。有些数据库驱动程序的确会在这一阶段对SQL语句做一些处理,另一些数据库驱动程序则只是简单地记住这条语句并等你用execute()方法去执行它们。prepare()方法的返回值是一个语句句柄,我们把它赋值给了变量\$sth。此后,只要是与这个语句有关的工作都要通过这个语句句柄来完成。

需要提醒大家注意的是,脚本程序里的查询语句并不需要用分号(;)来结尾。在与mysql客户程序打了那么长时间的交道之后,你可能已经习惯于在SQL语句的末尾加上一个“;”字符来作为它的结束符了。但在与DBI打交道的时候,还是把这个习惯改一改比较好,因为它往往会导致查询命令因语法错误而执行失败。这同样适用于查询命令末尾的“\g”结束符——最好不要再加上它了。这些语句结束符只有在mysql客户程序里才是必需的,通过DBI脚本发出的查询命令不需要使用它们。在DBI脚本里,查询命令字符串的尾字符就隐含地标志了查询命令的结束,用不着再由你去给它们画蛇添足地加上一个明确的结束符了。

如果在调用某个方法的时候没有给它传递任何输入参数,就可以省略它后面的括号。也就是说,下面两种写法是等价的:

```
$sth->execute ();
$sth->execute;
```

我喜欢留着那些括号,因为它们能帮我把脚本中的方法调用和变量引用区分开来。你的喜好也许和我不一样,但这并不是什么原则问题。

execute()调用相当于在mysql客户程序里执行有关的SQL语句。就dump\_members.pl脚本而言,在调用execute()之后,我们就可以开始对会员名单数据行进行处理了。在这个脚本里,用来对数据行进行处理的循环语句很简单,它只是把结果集中的每一个数据行的内容依次打印为

一组以制表符分隔的值而已：

```
# read results of query, then clean up
while (my @ary = $sth->fetchrow_array ())
{
    print join ("\t", @ary), "\n";
}
$sth->finish ();
```

fetchrow\_array()的返回值是由当前数据行各数据列的值构成的一个数组，如果已经到达结果集里的最后一个数据行，它就将返回一个空数组。于是，这个循环语句将依次取回SELECT语句返回的每一个数据行，在数据列值之间加上一些制表符，然后再把它们打印出来。数据库里的NULL值将被返回为Perl脚本里的undef值，但它们将被打印为空字符串而不是单词“NULL”。这些对应于数据库里的NULL值的undef值在Perl脚本里还有另外一个作用：在运行脚本时，它们会导致Perl解释器给出如下所示的警告信息：

```
Use of uninitialized value in join at dump_members.pl line n.
```

这些警告信息是因为在脚本程序的第一行使用了-w选项而产生的。如果去掉-w选项并再次运行这个脚本，这些警告信息就不会出现了。可是，-w选项能够帮助我们发现其他问题（比如试图打印一个未经初始化赋值的变量！），所以我们宁愿保留-w选项并另想一个办法来消除这些因undef值而产生的警告信息——将在第7.2.5节里介绍一些解决这一问题的办法。

在循环语句的print语句里，我们给制表符和换行符（它们被分别表示为“\t”和“\n”）都加上了双引号——因为Perl只能识别出放在双引号里的转义序列，而放在单引号里的转义序列将被当做普通的字符串来处理。如果使用的是单引号，这个脚本就会原封不动地打印出大量的“\t”和“\n”字符来。

我们在用来对数据行进行处理的循环语句的后面使用了一个finish()调用，它表示这个脚本对语句句柄\$sth的使用已经告一段落，现在可以释放当初为这个句柄而分配的各种临时资源了。就dump\_members.pl脚本而言，finish()调用只起到了一个演示的目的，实际上用不着在这个脚本里调用它，因为fetchrow\_array()调用已经在它到达结果集的末尾时替我们做过这件事情了。但在只取回并处理了结果集的一部分而没有到达其末尾的场合（比如因为某种原因而仅取回了结果集里的第一个数据行的时候），finish()就必不可少。在以后的示例脚本里，如果没有必要的话，将不使用finish()调用。

打印完会员名单，我们的工作也就结束了，所以要断开与服务器的连接并退出脚本：

```
$dbh->disconnect ();
exit (0);
```

虽然本身并不复杂，但dump\_members.pl脚本却给出了DBI程序设计的一个基本框架——即使你对DBI的了解只有这么多，也可以开始编写自己的DBI脚本了。比如说，如果只是想将其他数据表的内容也打印出来，那么只需在这个脚本里对prepare()方法里的SELECT语句做一下相应的修改就能达到目的。事实上，在第7.3节里就有很多脚本（比如用来生成“美国历史研究会”年会请柬和用来生成“美国历史研究会”排版格式会员名录的那两个脚本）是利用这里介绍的

基本技巧写出来的。不过,利用Perl脚本来执行简单的SELECT语句只是DBI向我们提供的众多功能中的一种,在后面的章节里还有更丰富的内容在等待着你们去学习。

### 7.2.3 出错处理

在调用connect()方法的时候,dump\_members.pl脚本激活了RaiseError出错处理属性,一旦执行出错,它就会在显示完出错信息之后自动结束这个脚本的运行而不是返回出错代码。其实还存在着一些其他的出错处理办法,你完全可以不用DBI代劳而自己去进行出错处理。

为了更好地了解DBI的出错处理机制,我们现在来仔细研究一下connect()调用的最后一个输入参数。这个参数涉及到了RaiseError和PrintError两个属性:

- 如果激活了RaiseError属性(即把它设置为一个非零值),那么,如果在某个DBI方法里出现了执行错误,DBI就会调用die()方法来显示出错信息并退出。
- 如果激活了PrintError属性(即把它设置为一个非零值),那么,如果在某个DBI方法里出现了执行错误,DBI就会调用warn()方法来显示一条出错信息却不会退出,脚本仍能继续往下执行。

在默认情况下,DBI会禁用RaiseError属性,激活PrintError属性。在这种情况下,如果connect()方法调用失败,DBI将显示一条出错信息,但仍会继续往下执行。也就是说,即使省略了connect()方法的第四个输入参数,DBI也能提供一种默认的出错处理行为。知道了这一点,就可以像下面这样来检查connect()方法的执行出错情况:

```
my $dbh = DBI->connect ($dsn, $user_name, $password)
    or exit (1);
```

如果真的在执行过程中出现了错误,connect()方法将返回一个undef以表明自己没有执行成功,而这个undef又将导致exit()被调用执行。用不着由你去打印一条出错信息,因为DBI已经打印出一条出错信息了。

如果想明确地把出错检查属性设置为它们的默认值,就应该把connect()调用写成下面这个样子:

```
my $dbh = DBI->connect ($dsn, $user_name, $password,
    { RaiseError => 0, PrintError => 1 })
    or exit (1);
```

虽说这会增加工作量,但它能让别人更清楚地了解你使用了什么样的出错处理行为。

如果想自己去完成出错处理工作并打印出一条个性化的出错信息,就应该把RaiseError和PrintError属性都禁用掉,如下所示:

```
my $dbh = DBI->connect ($dsn, $user_name, $password,
    { RaiseError => 0, PrintError => 0 })
    or die "Could not connect to server: $DBI::err ($DBI::errstr)\n";
```

上面这段代码里的\$DBI::err和\$DBI::errstr变量在你打算自行构造出错信息时会非常有用。它们分别包含着来自MySQL的出错代码和出错字符串,就像C API函数mysql\_errno()和mysql\_error()那样。如果没有发生错误,变量\$DBI::err将是0或undef,而变量\$DBI::errstr将是



空字符串或undef。换句话说,如果没有发生错误,这两个变量就都将是逻辑假值。

如果想让DBI替你去进行出错处理而不是由你本人去检查执行过程是否出现错误,就需要激活RaiseError属性并禁用PrintError属性,如下所示:

```
my $dbh = DBI->connect ($dsn, $user_name, $password,
                        { RaiseError => 1, PrintError => 0 });
```

这是一个最简单的办法,本章的大部分示例脚本也都是这样写的。之所以要在激活RaiseError属性的同时禁用PrintError属性,是为了避免出现可能会把同样原因的出错信息打印出来两次的情况。(在某些特定的场合,如果同时激活了这两个属性,DBI的出错处理机制就会因为同一个错误而打印出两条出错信息来。)

在某些场合,比如想让脚本程序在因为执行出错而半途退出之前做一些扫尾工作的时候,激活RaiseError属性的做法是不可取的。但对于这种情况,还可以采取另外一个办法来达到目的,即重新定义\$SIG{\_\_DIE\_\_}的处理方法。不激活RaiseError属性的另一个理由是DBI打印出来的出错信息专业性太强,不适合非技术人员。比如说:

```
disconnect(DBI::db=HASH(0x197aae4)) invalidates 1 active statement. Either
destroy statement handles or call finish on them before disconnecting.
```

这是一条对程序员非常有用的出错信息,但它可能是你想让普通用户们看到的东西。有时候,由自己来进行出错处理应该是一个更好的选择,因为这可以让使用脚本的人们看到一些更有实际意义的信息。对于这种情况,重新定义\$SIG{\_\_DIE\_\_}的处理方法也是一个值得考虑的解决方案。这种解决方案的好处是:一方面可以激活RaiseError属性以简化出错处理工作,另一方面又能把DBI默认提供的出错信息修改为自己的内容。如果采用重新定义\$SIG{\_\_DIE\_\_}的处理方法,请在执行任何一个DBI调用之前先做好下面这样的事情:

```
$SIG{__DIE__} = sub { die "Sorry, an error occurred\n"; };
```

也可以像普通情况那样先声明一个子例程,再把这个子例程设置为\$SIG{\_\_DIE\_\_}的处理方法。如下所示:

```
sub die_handler
{
    die "Sorry, an error occurred\n";
}

$SIG{__DIE__} = \&die_handler;
```

在dump\_members.pl脚本里,我们是把出错处理属性直接传递到connect()调用里去的。完全可以先把它们设置为一个散列变量,然后再把这个变量的引用指针传递到connect()调用里去。从效果上讲,这两种办法其实是一样的,但有些人觉得把属性的设置和使用分开进行的办法能够让脚本代码更容易阅读和修改。下面这个例子演示了使用一个散列变量来传递有关属性的具体做法:

```
my %attr =
(
    PrintError => 0,
```



```

        RaiseError => 0
    );
    my $dbh = DBI->connect ($dsn, $user_name, $password, \%attr)
        or die "Could not connect to server: $DBI::err ($DBI::errstr)\n";

```

下面这个dump\_members2.pl脚本演示了由你来负责出错处理工作并提供个性化出错信息的情况。dump\_members2.pl脚本将要处理的查询命令与dump\_members.pl脚本里的一模一样,但它明确地禁用了RaiseError和PrintError属性,其中每一个DBI调用的结果都将由脚本代码来负责检查。一旦执行出错,这个脚本会在退出之前先调用子例程bail\_out()来打印一条相应的出错信息以及\$DBI::err和\$DBI::errstr变量的内容:

```

#! /usr/bin/perl -w
# dump_members2.pl - dump Historical League's membership list

use strict;
use DBI;

my $dsn = "DBI:mysql:sampdb:cobra.snake.net";    # data source name
my $user_name = "sampadm";                      # user name
my $password = "secret";                        # password
my %attr =                                       # error-handling attributes
(
    PrintError => 0,
    RaiseError => 0
);

# connect to database
my $dbh = DBI->connect ($dsn, $user_name, $password, \%attr)
    or bail_out ("Cannot connect to database");

# issue query
my $sth = $dbh->prepare ("SELECT last_name, first_name, suffix, email,"
    . "street, city, state, zip, phone FROM member ORDER BY last_name")
    or bail_out ("Cannot prepare query");
$sth->execute ()
    or bail_out ("Cannot execute query");

# read results of query
while (my @ary = $sth->fetchrow_array ())
{
    print join ("\t", @ary), "\n";
}
!$DBI::err
    or bail_out ("Error during retrieval");

$dbh->disconnect ()
    or bail_out ("Cannot disconnect from database");

```

```

exit (0);

# bail_out() subroutine - print error code and string, then exit

sub bail_out
{
    my $message = shift;

    die "$message\nError $DBI::err ($DBI::errstr)\n";
}

```

子例程**bail\_out()**与我们在第6章里学习编写C语言程序时使用的**print\_error()**函数很相似，但**bail\_out()**会在打印完有关信息之后立刻退出执行而不是返回到它的调用者那里。子例程**bail\_out()**给我们带来的好处有两个：首先，在打印一条出错信息的时候，虽然没有把它们写出来，但**\$DBI::err**和**\$DBI::errstr**变量的内容也会自动地跟在个性化出错信息的后面被打印出来；其次，因为把出错信息的打印功能封装在了一个子例程里，所以只需简单地修改一下这个子例程，就可以让整个脚本里的出错信息格式发生统一的变化。

在**dump\_members2.pl**脚本里，我们在用来取回和处理数据行的循环语句后面安排了一个检测机制。这是因为这个脚本不会在**fetchrow\_array()**调用执行出错的时候自动退出，所以有必要安排一个这样的检测机制来检查整个循环是因为到达结果集的末尾而结束（正常结束）还是因为在执行过程中发生了错误而结束（非正常结束）：如果循环是正常结束，这个脚本的输出结果将是一份完整的会员名单；如果循环是非正常结束，这个脚本的输出结果就不是一份完整的会员名单。虽然循环在两种情况下都会结束，但如果没有这个检测机制，我们就无法知道这个脚本是否“偷工减料”，所看到的会员名单是不是完整。总而言之，如果想自己去进行出错处理，千万不要忘记在用来取回结果集的循环语句后面加上一个这样的检测机制。

#### 7.2.4 处理没有结果集的查询

有些SQL语句（如**SELECT**、**DESCRIBE**、**EXPLAIN**、**SHOW**等）会返回一些数据行，另外一些语句（如**DELETE**、**INSERT**、**REPLACE**、**UPDATE**等）则不会返回数据行。与前一类语句相比，后一类语句的处理要相对容易一些。对于非**SELECT**语句，可以用数据库句柄把它们传递到DBI脚本的**do()**方法里去处理。**do()**方法能够把SQL查询命令的准备和执行工作合并为一个步骤去完成。比如说，下面的脚本代码将把一个姓名为**Marcia Brown**，会员资格失效日期为2005年6月3日的新记录插入到**member**数据表里去：

```

$rows = $dbh->do ("INSERT INTO member (last_name,first_name,expiration)"
    . " VALUES('Brown','Marcia','2005-6-3')");

```

**do()**方法的返回值是受其影响的数据行的个数，若执行出现错误，则返回**undef**；若因为某种原因无法确定受其影响的数据行个数，则返回-1。有很多原因都会造成**do()**方法的执行出错，比如查询命令本身有语法错误或者拥有的权限不允许你访问某个数据表等。对于**do()**方法的非**undef**返回值，应该检查受其影响的数据行个数是不是0。值得注意的是，当真的出现受其影响

的数据行个数等于0的情况时, do()方法的返回值并不是数字0, 它将返回一个字符串"0E0" (即数字0在Perl语言里的科学记数法表示形式)。在数值上下文里, "0E0"将被求值为0; 但在条件表达式里, 它却会被求值为真。这使我们很容易把它与undef区别开。假如do()返回的是数字0的话, 就很难把“发生了错误”(undef)和“没有数据行受到它的影响”这两种情况区别开了。下面两个测试都能用来判断执行过程是否出现了错误:

```
if (!defined ($rows))
{
    print "An error occurred\n";
}
if (!$rows)
{
    print "An error occurred\n";
}
```

在数值上下文里, "0E0"将被求值为0, 所以下面这段代码能够根据非undef的\$rows值正确地计算并打印出数据行的个数:

```
if (!$rows)
{
    print "An error occurred\n";
}
else
{
    $rows += 0; # force conversion to number if value is "0E0"
    print "$rows rows affected\n";
}
```

还可以用printf()函数的“%d”格式符来打印\$rows, 这将把字符串"0E0"强制转换为一个数字:

```
if (!$rows)
{
    print "An error occurred\n";
}
else
{
    printf "%d rows affected\n", $rows;
}
```

do()方法等价于prepare()加execute()。这也就是说, 前面那句用do()方法来处理INSERT语句的脚本代码与下面的代码是等价的:

```
$sth = $dbh->prepare ("INSERT INTO member (last_name,first_name,expiration)"
    . " VALUES('Brown','Marcia','2005-6-3')");
$rows = $sth->execute ();
```

### 7.2.5 处理有结果集的查询

在DBI脚本里, 我们必须通过一个循环来取回和处理SELECT查询 (或者像SELECT这样能

够返回一些数据行的其他语句，如DESCRIBE、EXPLAIN、SHOW等)。在这一小节里，将向大家介绍这类循环结构的几种编程方法。此外，还将学习如何查知结果集里的数据行个数、在只返回有一个数据行的场合（即不需要使用循环结构的时候）如何对结果集进行处理、如何把结果集当做一个整体来处理等内容。

#### 1. 用循环语句来取回结果集中的数据行

dump\_members.pl脚本使用了一系列标准的DBI方法来检索数据：用prepare()方法对查询命令进行预处理，用execute()方法开始执行查询命令，再用fetchrow\_array()方法依次取回结果集里的每一个数据行并对之进行处理。

在准备对有结果集的查询命令进行处理的脚本里，用来完成预处理工作的prepare()和用来完成执行工作的execute()都是相当标准的组成部分。但可以用来取回数据行的DBI方法却有好几种（见表7-3），我们前面见过的fetchrow\_array()只是其中之一而已。

表7-3 可以用来取回数据行的DBI方法

方法名称	返回值
fetchrow_array()	一个数组，其元素是结果集里的数据行的值
fetchrow_arrayref()	一个引用指针，指向一个由结果集里全体数据行的值构成的数组
fetchrow()	与fetchrow_arrayref()相同
fetchrow_hashref()	一个引用指针，指向一个散列表，散列表里的元素由结果集里的数据行的值构成，键字是数据列名

下面的示例演示了表7-3列出的各个方法的用法。这些示例将通过一个循环结构来取回某结果集里的每一个数据行，并把数据列的值以逗号间隔打印出来。有些示例本来是可以编写得更有效的，但因为这里的目的是为了演示这些DBI方法的用法（即它们用来访问各数据列的值时所使用的编程语法）而不是追求执行效率，所以没有那样做。

我们先来看看fetchrow\_array()的用法：

```
while (my @ary = $sth->fetchrow_array ())
{
    my $delim = "";
    for (my $i = 0; $i < @ary; $i++)
    {
        $ary[$i] = "" if !defined ($ary[$i]);    # NULL value?
        print $delim . $ary[$i];
        $delim = ",";
    }
    print "\n";
}
```

每调用fetchrow\_array()一次，它就返回一个数组，数组元素是结果集里的数据行的值；若已经到达结果集里的最后一个数据行，则返回一个空数组。内层循环将依次检查各数据列的值是否有定义（即是否为undef），如果没有定义，就把它设置为空字符串。这将把数据库里的NULL值（它们在DBI脚本里被表示为undef）全部转换为空字符串。这项工作乍看起来好像有

点多余——因为不管是undef还是空字符串, Perl都将把它们打印成空白。之所以要做这样的测试和转换, 是因为如果脚本运行在有-w选项的严格模式下, Perl就会在你试图打印一个undef值的时候给出一条“Use of uninitialized value”(使用了未经初始化的值)警告信息。在本章的很多示例里, 都能看到这种为预防万一而编写出来的代码。如果想把undef打印为另外一个值, 比如单词“NULL”, 只要稍微修改一下测试部分的代码就可以了, 如下所示:

```
while (my @ary = $sth->fetchrow_array ())
{
    my $delim = "";
    for (my $i = 0; $i < @ary; $i++)
    {
        $ary[$i] = "NULL" if !defined ($ary[$i]); # NULL value?
        print $delim . $ary[$i];
        $delim = ",";
    }
    print "\n";
}
```

既然是对一个数组进行处理, 我们就可以利用Perl语言中的map操作符来简化所编写出来的代码, 把数组里取值为undef的元素一次性地全部转换过来, 如下所示:

```
while (my @ary = $sth->fetchrow_array ())
{
    @ary = map { defined ($_) ? $_ : "NULL" } @ary;
    print join (",", @ary) . "\n";
}
```

map操作符将根据花括号里的表达式依次对数组中的每一个元素进行检查和转换(如果需要的话), 最后返回一个由转换结果值构成的数组。

上面这一系列做法的整体思路是把fetchrow\_array()的返回值赋值给一个数组变量。除这个办法外, 还可以把各数据列的值取到一组标量变量里来, 而这将使我们能够用一个比较有意义的变量名来代替没有明显意义的\$array[0]、\$array[1]之类的东西。我们来看一个例子。假设有会员的姓名和电子邮件地址检索到相应的变量里去, 使用fetchrow\_array()方法的代码如下所示:

```
my $sth = $dbh->prepare ("SELECT last_name, first_name, suffix, email"
                        . " FROM member ORDER BY last_name");
$sth->execute ();
while (my ($last_name, $first_name, $suffix, $email) = $sth->fetchrow_array ())
{
    # do something with variables
}
```

当准备像上面这样使用一组变量来取回各数据列的值的时候, 一定要保证在SQL查询命令中给出的数据列名的顺序与用来保存有关数据的那组变量的排列顺序相符合。DBI并不清楚在SELECT语句里是按什么样的顺序来列出数据列名的, 所以给变量正确赋值的责任就落在了你的身上。利用一种称为“参数绑定”的技巧, 我们还可以使数据列的值被自动赋值给各个变量。



将在第7.2.7节对这一技巧做进一步的介绍。

如果准备把各数据列的值分别取回到一组标量变量里，一定要在对变量进行赋值的时候多加小心。如果循环是以下面这行代码开头的，它将正确工作：

```
while (my ($val) = $sth->fetchrow->array ()) ...
```

这是一个列表型上下文，所以有关数据将被正确地取回并赋值给列表变量\$val，只有在已经到达结果集里的最后一个数据行时，while循环的条件表达式才会被求值为假。但如果把while循环的条件表达式写成了下面这个样子（缺少了括号），就会出现一些很奇怪的问题：

```
while (my $val = $sth->fetchrow->array ()) ...
```

这是一个标量型上下文，如果标量变量\$val的取值恰好是0、undef或空字符串的话，那么即使还没有到达结果集里的最后一个数据行，while语句也会因其条件表达式被求值为假而结束循环。

表7-3里的第二个方法fetchrow\_arrayref()，与我们刚才介绍的fetchrow\_array()很相似，但它返回的不是一个由当前数据行里的各个数据列的值所构成的数组，而是一个指向该数组的引用指针，若已经到达结果集里的最后一个数据行，则返回undef。下面是它的使用方法：

```
while (my $ary_ref = $sth->fetchrow_arrayref ())
{
    my $delim = "";
    for (my $i = 0; $i < @{$ary_ref}; $i++)
    {
        $ary_ref->[$i] = "" if !defined ($ary_ref->[$i]); # NULL value?
        print $delim . $ary_ref->[$i];
        $delim = ",";
    }
    print "\n";
}
```

在这段代码里，必须通过数组引用指针\$ary\_ref来访问数组中的各个元素。这相当于C语言中对指针进行“退指针化处理”（即通过指针来访问该指针所指向的数据）的效果，所以必须把它写成“\$ary\_ref->[\$i]”的样子，而不能写成“\$ary[\$i]”的样子。如果想把这个数组引用指针转换为一个数组，可以使用Perl语言提供的“@{\$ary\_ref}”构造。

fetchrow\_arrayref()不适合用来把有关数据取回到一组标量变量里去。比如说，下面这个样子的循环是无法正确工作的：

```
while (my ($var1, $var2, $var3, $var4) = @{$sth->fetchrow_arrayref ()})
{
    # do something with variables
}
```

在上面这个循环里，如果fetchrow\_arrayref()真的取回了一个数据行，循环尚可以继续执行。可一旦到达结果集里的最后一个数据行，fetchrow\_arrayref()就将返回undef，而@{undef}却是非法的（它类似于在C语言里对一个NULL指针进行退指针化处理的情况）。

表7-3里的第四个方法fetchrow\_hashref()的使用情况如下所示:

```
while (my $hash_ref = $sth->fetchrow_hashref ())
{
    my $delim = "";
    foreach my $key (keys (%{$hash_ref}))
    {
        $hash_ref->{$key} = "" if !defined ($hash_ref->{$key}); # NULL value?
        print $delim . $hash_ref->{$key};
        $delim = ",";
    }
    print "\n";
}
```

每调用fetchrow\_hashref()一次,它就返回一个指向一个散列列表的引用指针,散列列表里的元素由结果集里的某一个数据行构成,键字是数据列名;若已经到达结果集里的最后一个数据行,则返回undef。需要提醒大家注意的是,由fetchrow\_hashref()取回的数据列的值没有什么排列顺序可言,这是因为Perl对散列列表里的元素不进行任何排序。好在DBI会把各数据列名用做各散列元素的键字,这意味着我们完全能够通过仅有的那个\$hash\_ref变量根据某数据列名(即散列元素的键字)把该数据列的取值(即哈希元素的键值)给找出来。这同时还意味着我们完全可以按任意顺序来提取某一个(或者某几个)数据列的取值而不必理会这些数据列在SELECT查询里的检索顺序到底是怎样的。比如说,如果想提取出某个会员的姓名和电子邮件地址,只需写出下面这样的代码就行了:

```
while (my $hash_ref = $sth->fetchrow_hashref ())
{
    my $delim = "";
    foreach my $key ("last_name", "first_name", "suffix", "email")
    {
        $hash_ref->{$key} = "" if !defined ($hash_ref->{$key}); # NULL value?
        print $delim . $hash_ref->{$key};
        $delim = ",";
    }
    print "\n";
}
```

fetchrow\_hashref()特别适用于这样的场合:需要把一个数据行的值传递给一个函数,但这个函数却不必知道有关的数据列在SELECT语句里的先后顺序。具体做法是:使用fetchrow\_hashref()方法依次取回各数据行的值,再编写一个函数并让它使用各有关数据列名从由fetchrow\_hashref()方法返回的数据行散列列表里取出相应的值。

在使用fetchrow\_hashref()方法的时候,请注意以下几个问题:

- 如果想获得最佳的执行性能,那么fetchrow\_hashref()就不是最佳的选择。它的执行效率要比fetchrow\_array()和fetchrow\_arrayref()低。
- 在默认的情况下,充当散列键字的数据列名将沿用它们在SELECT语句里的大小写情况。在MySQL里,数据列名是不区分字母大小写情况的,所以不管使用什么样的字母大小写

组合来书写数据列名，查询命令都能正确执行。但Perl的散列键字却是要区分字母大小写情况的，这就有可能导致一些问题。为了避免这种因字母大小写不匹配而造成的意外，可以给fetchrow\_hashref()方法传递一个NAME\_lc或NAME\_uc属性来强制它把数据列名统一为一致的大小写情况，如下所示：

```
$hash_ref = $sth->fetchrow_hashref ("NAME_lc"); # use lowercase names
$hash_ref = $sth->fetchrow_hashref ("NAME_uc"); # use uppercase names
```

- 散列列表中的每个元素必须对应着一个独一无二的数据列名。如果同时对多个数据表进行了关联查找而这些数据表彼此又有重复出现的数据列名，就无法取回全部的数据列值。比如说，如果发出了一个下面这样的查询，fetchrow\_hashref()就将返回一个只包含有name这一个元素的散列列表：

```
SELECT a.name, b.name FROM a, b WHERE a.name = b.name
```

为了避免这种问题，应该增加一些别名来保证每个数据列都有一个独一无二的名字。比如说，如果把刚才那个查询改写为下面的样子，就能让fetchrow\_hashref()返回一个包含有name和name2两个元素的散列列表的引用指针了：

```
SELECT a.name, b.name AS name2 FROM a, b WHERE a.name = b.name
```

## 2. 查知某个查询返回了多少个数据行

怎样才能知道所发出的SELECT查询（或者相似于SELECT的查询）到底返回了多少个数据行呢？一种办法是一边取回结果集里的数据行，一边对它们进行计数。事实上，这是DBI里惟一能够做到既能查知SELECT查询到底返回了多少个数据行，又具备可移植性的手段。单就MySQL而言，还可以在调用完execute()之后立刻用刚才的语从句柄去调用row()方法。但这个办法不能移植到其他数据库引擎里去使用，DBI的官方文档里也明确地不鼓励使用row()方法去查知SELECT语句到底返回了多少个数据行。而且，即便是在MySQL里，如果设置了mysql\_use\_result属性，那么在取回结果集里的全部数据行之前，row()返回的统计数字也不可靠。因此，最保险的办法还是一边取回结果集里的数据行，一边对它们进行计数。（有关mysql\_use\_result属性的详细说明请参见附录G。）

## 3. 取回只有一个数据行的查询结果

如果事先就已经知道结果集里只有一个数据行，就没必要使用循环结构来取回它。比如说，如果想编写一个名为count\_members.pl的脚本来统计“美国历史研究会”现在到底有多少名会员，就可以用下面这样的代码来完成有关的查询：

```
# issue query
my $sth = $dbh->prepare ("SELECT COUNT(*) FROM member");
$sth->execute ();

# read results of query
my $count = $sth->fetchrow_array ();
$sth->finish ();
$count = "can't tell" if !defined ($count);
print "$count\n";
```

在上面这段代码里, SELECT语句将只返回一个数据行, 所以就没有必要去使用循环结构, 只需调用一次fetchrow\_array()方法就足够了。同时, 因为我们只选取了一个数据列, 所以甚至不必把fetchrow\_array()方法的返回值赋值给一个数组。当在一个标量型上下文里调用fetchrow\_array()方法(即已经确知它只会返回一个值而不是一组值)的时候, 它将只返回中选数据行的某一个数据列; 若没有数据行中选, 则返回undef。至于fetchrow\_array()在标量型上下文里将会具体返回中选数据行的哪一个数据列, DBI并没有做出明确的规定。但这对上面的查询没有丝毫影响。这段代码只检索出了一个值, 不可能产生二义性。

虽然结果集里只有一个数据行, 但这段代码还是调用了finish()方法来释放临时分配的各种资源。(在到达结果集里最后一个数据行的时候, fetchrow\_array()方法将隐含地释放为它临时分配的各种资源。但就这个示例而言, 只有当第二次调用它时才会有这样的效果。)

还有一种查询(即给出了LIMIT 1子句的那些查询)也是结果集里最多也只会会有一个数据行的。这种查询经常被用来查找某个数据列里的最大值或者最小值。比如说, 下面这段代码将把出生日期距今最近的那位美国总统的姓名和出生日期查找并打印出来:

```
my $query = "SELECT last_name, first_name, birth"
           . " FROM president ORDER BY birth DESC LIMIT 1";
my $sth = $dbh->prepare ($query);
$sth->execute ();

# read results of query
my ($last_name, $first_name, $birth) = $sth->fetchrow_array ();
$sth->finish ();
if (!defined ($last_name))
{
    print "Query returned no result\n";
}
else
{
    print "Most recently born president: $first_name $last_name ($birth)\n";
}
```

使用了MAX()或MIN()函数的查询命令也将只返回一个数据行, 所以也不需要使用循环结构来取回它们结果集里的数据行。DBI为上面提到的这几种只返回一个数据行的情况特意准备了一个更加简便的处理手段, 即通过数据库句柄来调用selectrow\_array()方法——这等于是把prepare()、execute()、数据行取回操作等多项工作合并到一个调用步骤里来了。如果执行成功, 这个方法将返回一个数组(注意: 不是一个引用指针); 如果查询没有返回数据行或者在执行过程中出现了错误, 它将返回一个空数组。上面那段示例代码可以用selectrow\_array()方法改写为如下所示的样子:

```
my $query = "SELECT last_name, first_name, birth"
           . " FROM president ORDER BY birth DESC LIMIT 1";
my ($last_name, $first_name, $birth) = $dbh->selectrow_array ($query);
if (!defined ($last_name))
{
```

```

    print "Query returned no result\n";
}
else
{
    print "Most recently born president: $first_name $last_name ($birth)\n";
}

```

#### 4. 对结果集进行整体处理

如果使用循环结构来取回结果集里的数据行，那么，除了按数据行在结果集里的先后顺序来依次对它们进行处理之外，DBI没有提供任何能让你按其他顺序来处理它们的手段。而且，如果没有采取适当的措施，在取回下一个数据行之后，上一个数据行就会“丢失”在内存里。这些行为并不总是我们所希望的。比如说，当需要对结果集里的数据行做多次遍历以完成某项统计工作（比如说，第一遍是对有关数据的粗略分析，而第二遍才是正式的细致统计）时，你肯定不希望发生这种“狗熊掰棒子”的事情。

其实，我们还是有机会把结果集当做一个整体来访问的。首先，可以像平常一样使用循环结构来依次取回结果集里的数据行，但每取回一个数据行，就立刻把它保存起来。其次，有些DBI方法能让你只进行一次调用就把整个结果集都取回来。不管采取的是哪一种策略，最终得到的都将是一个这样的矩阵：它的每一行分别对应着结果集里的一个数据行，而它的每一列则分别对应着你在SQL语句里给出的一个数据列。只要构造出了这个矩阵，想按什么次序来处理其中的元素、想处理多少次就都不是问题了。下面，我们将对这两种策略分别进行讨论。

先说第一种策略——用循环结构来构造结果集矩阵。其具体思路是：每用`fetchrow_array()`方法取回一个数据行，就把该数据行的引用指针保存到一个数组里。下面这段代码的执行效果与`dump_members.pl`脚本里的“取回+打印”循环差不多，但这里的做法是先把全体数据行保存到一个矩阵里，然后再把那个矩阵打印出来。这段代码有两个地方请大家特别留意：它是如何确定矩阵里有多少个行和列的；它是如何去访问矩阵里的各个元素的。

```

my @matrix = (); # array of array references

while (my @ary = $sth->fetchrow_array ()) # fetch each row
{
    push (@matrix, [ @ary ]); # save reference to just-fetched row
}

# determine dimensions of matrix
my $rows = scalar (@matrix);
my $cols = ($rows == 0 ? 0 : scalar (@{$matrix[0]}));

for (my $i = 0; $i < $rows; $i++) # print each row
{
    my $delim = "";
    for (my $j = 0; $j < $cols; $j++)
    {
        $matrix[$i][$j] = "" if !defined ($matrix[$i][$j]); # NULL value?
        print $delim . $matrix[$i][$j];
    }
}

```



```

        $delim = ",";
    }
    print "\n";
}

```

在确定矩阵行列数的时候, 必须先把它的行数确定下来——因为是否需要确定矩阵的列数首先要看它是否为空: 如果行数\$row等于0, 即矩阵是空的, 那它的列数\$col也将等于0; 只有在\$row不等于0的时候, 才有必要去确定\$col的值。因为矩阵的列数就等于其第一行里的元素个数, 所以, 在知道了矩阵的行数之后, 我们就能用语法@{\$matrix[0]}把它的列数确定下来。

上面这段示例代码是这样构造结果集矩阵的: 把结果集里的每一个数据行依次取回为一个数组并把它的一个引用指针保存到了矩阵里。你也许会认为像下面这样用fetchrow\_arrayref()方法来直接检索数据行引用指针的做法会更有效:

```

my @matrix = (); # array of array references

while (my $ary_ref = $sth->fetchrow_arrayref ())
{
    push (@matrix, $ary_ref); # save reference to just-fetched row
}

```

只可惜这种做法是行不通的。这是因为fetchrow\_arrayref()会反复地使用同一个数组来取回各个数据行, 换句话说, 那些引用指针都将指向同一个数组。你虽然会得到一个以引用指针为元素的矩阵, 可它们却全都指向同一个数据行——结果集里的最后一个数据行。因此, 如果想用一次取回一个数据行的办法来构造结果集矩阵, 就一定要使用fetchrow\_array()方法, 而不要使用fetchrow\_arrayref()方法。

下面再来看看第二种策略——利用某个能让你只进行一次调用就把整个结果集都取回来的DBI方法。比如说, fetchall\_arrayref()方法将返回一个引用指针, 它指向一个元素为引用指针的数组, 每一个数组元素(引用指针)分别指向一个由结果集里的某个数据行的内容构成的数组。虽然听起来很复杂, 但从实际效果上讲, 这个方法的返回值恰好就是结果集矩阵的引用指针。fetchall\_arrayref()的具体用法是: 先依次调用prepare()和execute(), 再用下面这样的代码把整个结果集一次性地取回来:

```

# fetch all rows into a reference to an array of references
my $matrix_ref = $sth->fetchall_arrayref ();

```

下面这段代码将完成确定结果集矩阵的行列数以及访问各矩阵元素的工作:

```

# determine dimensions of matrix
my $rows = (defined ($matrix_ref) ? 0 : scalar (@{$matrix_ref}));
my $cols = ($rows == 0 ? 0 : scalar (@{$matrix_ref->[0]}));

for (my $i = 0; $i < $rows; $i++) # print each row
{
    my $delim = "";
    for (my $j = 0; $j < $cols; $j++)
    {

```

```

        $matrix_ref->[$i][$j] = "" if !defined ($matrix_ref->[$i][$j]); # NULL?
        print $delim . $matrix_ref->[$i][$j];
        $delim = ",";
    }
    print "\n";
}

```

若结果集为空，fetchall\_arrayref()将返回一个指向空数组的引用指针；若执行出错，则返回undef。因此，在使用这个方法的时候，不必激活它的RaiseError属性，但在进行后续操作之前，必须先检查它的返回值。

在确定结果集矩阵的行列数之前，一定要先检查它是否为空。如果想把这个矩阵的某一行（比如\$i）整个地当做一个数组来访问，就需要使用语法@{\$matrix\_ref->[\$i]}来进行。

很明显，利用fetchall\_arrayref()来检索结果集的做法要比使用循环结构的做法更简单易行，只是用来访问矩阵元素的语法稍微古怪了点。DBI还提供了一个与fetchall\_arrayref()相类似却又比它更省事的selectall\_arrayref()方法，这个方法等于是把prepare()、execute()、数据行取回循环等一整套流程合并到了一个调用步骤里。如果打算使用selectall\_arrayref()方法，只需把SQL查询命令直接通过数据库句柄传递给它就行了，如下所示：

```

# fetch all rows into a reference to an array of references
my $matrix_ref = $dbh->selectall_arrayref ($query);

# determine dimensions of matrix
my $rows = (!defined ($matrix_ref) ? 0 : scalar (@{$matrix_ref}));
my $cols = ($rows == 0 ? 0 : scalar (@{$matrix_ref->[0]}));

for (my $i = 0; $i < $rows; $i++)          # print each row
{
    my $delim = "";
    for (my $j = 0; $j < $cols; $j++)
    {
        $matrix_ref->[$i][$j] = "" if !defined ($matrix_ref->[$i][$j]); # NULL?
        print $delim . $matrix_ref->[$i][$j];
        $delim = ",";
    }
    print "\n";
}

```

## 5. NULL值的检测与处理

在从数据库检索信息的时候，我们经常需要把数据列里的NULL值与数值0或空字符串区分开来。因为DBI会把数据库里的NULL值返回为undef，所以这做起来并不困难，但必须按正确的顺序来进行测试才能真正解决问题。就拿下面这段示例代码来说，它的三条print语句打印出来的全都是“false!”：

```

$col_val = undef; if (! $col_val) { print "false!\n"; }
$col_val = 0;     if (! $col_val) { print "false!\n"; }
$col_val = "";    if (! $col_val) { print "false!\n"; }

```

这表明以上测试无法正确区分undef、数值0和空字符串。下面这段代码将打印出两个“false!”来,从而表明有关测试无法正确区分undef和空字符串:

```
$col_val = undef; if ($col_val eq "") { print "false!\n"; }
$col_val = "";   if ($col_val eq "") { print "false!\n"; }
```

下面这段代码的输出结果还是两个“false!”,说明第二个测试不能把数值0和空字符串区分开来。

```
$col_val = "";
if ($col_val eq "") { print "false!\n"; }
if ($col_val == 0)  { print "false!\n"; }
```

要想把数据列里的undef (NULL值) 和非undef区分开来,就必须使用defined()方法。只有先判定了数据列里的某个值不是NULL之后,才有必要通过进一步的测试把它与其他类型的“空”值区分开来。比如说:

```
if (!defined ($col_val)) { print "NULL\n"; }
elsif ($col_val eq "")   { print "empty string\n"; }
elsif ($col_val == 0)    { print "zero\n"; }
else                     { print "other\n"; }
```

注意,上面这些测试的顺序非常重要:如果\$col\_val是一个空字符串的话,第二和第三个比较操作就都将为真。换句话说,如果把这两个测试的顺序弄颠倒了的话,就不能正确地把空字符串和数值0区分开来。

### 7.2.6 引号问题

到目前为止,我们在DBI脚本里构造出来的SQL查询命令还全都是一些简单地加上了引号的字符串。我们应该庆幸至今仍未遇到什么麻烦,因为假如这些放在引号中的字符串里有带引号的值的话,它们早就在Perl语言的词法分析层面上引起一片混乱了。不仅如此,假如我们曾经试图插入或者选取过一些包含有引号、反斜线或者二进制数据的值,那也早就在SQL层面上遇到一些麻烦了。在把一条查询命令构造为一个Perl语言里的带引号字符串时,必须对查询命令字符串里面的每一个引号类字符(单引号、双引号、反斜线等等)进行转义。如下所示:

```
$query = 'INSERT INTO absence VALUES(14,\''2002-9-16'\')';
$query = "INSERT INTO absence VALUES(14,\"2002-9-16\")";
```

Perl和MySQL里的字符串都是既可以放在单引号里,也可以放在双引号里。利用这一特点,有时可以通过混用单、双引号的办法来避免对有关字符进行转义:

```
$query = 'INSERT INTO absence VALUES(14,"2002-9-16")';
$query = "INSERT INTO absence VALUES(14,'2002-9-16')";
```

可是,这两种引号在Perl语言里是不等效的——只有双引号里的变量名才能被解释为变量。因此,如果想在查询命令字符串里使用变量,就不能用单引号来构造查询命令。比如说,下面两个字符串就不是等价的(假设变量\$var的值是14):

```
"SELECT * FROM member WHERE id = $var"
'SELECT * FROM member WHERE id = $var'
```

下面是Perl对这两个字符串的解释，很明显，想传递给MySQL服务器的查询命令应该是第一个字符串：

```
"SELECT * FROM member WHERE id = 14"
'SELECT * FROM member WHERE id = $var'
```

除把字符串放在双引号中的办法外，还可以使用qq{}构造——Perl将把“qq{”和“}”之间的东西视为一个放在双引号中的字符串。比如说，下面两行代码是等价的：

```
$date = "2002-9-16";
$date = qq{2002-9-16};
```

qq{}构造既允许在查询命令字符串里随意使用引号（单引号或双引号）而不必对它们进行转义，也能对变量引用做出正确的解释，所以只要在构造查询命令时使用了qq{}，就用不着再顾虑引号的使用是否正确了。qq{}构造的这两个特点在下面的INSERT语句里得到了充分的体现：

```
$id = 14;
$date = "2002-9-16";
$query = qq{INSERT INTO absence VALUES($id,$date)};
```

qq构造的分隔符并非只能使用“{”和“}”，完全可以把它写成“qq()”或“qq//”的样子，但这样做的前提是“()”或“\\”不会出现在有关的字符串里。我个人比较喜欢使用“qq{}”——因为字符“}”在查询命令字符串里出现的机会非常少，不容易被误认为是查询命令的结束字符。而字符“)”和“\”就不同了。就拿“)”来说，它在每一条INSERT语句里几乎都会出现，所以用“qq()”来构造查询命令就很容易引起问题。

qq{}构造允许跨越多个代码行，这使我们得以把SQL查询命令字符串在Perl代码里凸现出来，如下所示：

```
$id = 14;
$date = "2002-9-16";
$query = qq{
    INSERT INTO absence VALUES($id,$date)
};
```

我们还可以利用这一特点把查询命令写在多个代码行上以增加它的可读性。比如说，在dump\_members.pl里使用了一条下面这样的SELECT语句：

```
$sth = $dbh->prepare ("SELECT last_name, first_name, suffix, email,"
    . "street, city, state, zip, phone FROM member ORDER BY last_name");
```

利用qq{}构造允许跨越多个代码行的特点，我们可以把它改写为下面这个样子：

```
$sth = $dbh->prepare (qq{
    SELECT
        last_name, first_name, suffix, email,
        street, city, state, zip, phone .
```

```

FROM member
ORDER BY last_name
));

```

虽说放在双引号中的字符串也允许跨越多个代码行,但我认为“qq{”和“}”要比两个孤零零的“”字符更抢眼,能使查询命令更容易阅读。这两种格式在本书里都有使用,至于哪一种更好,就请大家自己去判断吧。

qq{}构造解决了Perl词法层面上的引号问题,它使我们能够在字符串里随意使用引号而不会引起Perl的抱怨。但我们还必须更进一步地考虑到SQL层面上的语法问题。请看下面这段代码,它试图把一条新记录插入到member数据表里去:

```

$last = "O'Malley";
$first = "Brian";
$expiration = "2005-9-1";
$rows = $dbh->do (qq{
    INSERT INTO member (last_name,first_name,expiration)
    VALUES('$last','$first','$expiration')
});

```

这里的do()方法发送给MySQL服务器的查询命令字符串将是下面这个样子:

```

INSERT INTO member (last_name,first_name,expiration)
VALUES('O'Malley','Brian','2005-9-1')

```

这不是合法的SQL命令,因为在单引号里的字符串里又出现了单引号字符。在第6章里也曾遇到过类似的问题,我们当时是用mysql\_real\_escape\_string()函数来解决这一问题的。DBI也提供了一个类似的机制——当需要在查询命令里使用一个带引号的值时,可以把它传递给quote()方法,然后把这个方法的返回值用在查询命令里。就拿上面那个例子来说,把它写成下面这样就不会有问题了:

```

$last = $dbh->quote ("O'Malley");
$first = $dbh->quote ("Brian");
$expiration = $dbh->quote ("2005-9-1");
$rows = $dbh->do (qq{
    INSERT INTO member (last_name,first_name,expiration)
    VALUES($last,$first,$expiration)
});

```

现在,do()方法发送给MySQL服务器的查询命令字符串将是下面这个样子——带引号的字符串里的引号字符都得到了正确的转义处理:

```

INSERT INTO member (last_name,first_name,expiration)
VALUES('O\'Malley','Brian','2005-9-1')

```

需要注意的是,查询命令字符串里的\$last和\$first变量都不应该再用引号把它们引起来了,这是因为quote()方法会替你给它们加上必要的引号。如果你还画蛇添足地给它们加上了引号,查询命令字符串里的引号就会太多了,就像下面这个例子演示的那样:



```

$value = "paul";
$quoted_value = $dbh->quote ($value);

print "... WHERE name = $quoted_value\n";
print "... WHERE name = '$quoted_value'\n";

```

这些语句将产生如下所示的输出：

```

... WHERE name = 'paul'
... WHERE name = ''paul''

```

很明显，第二个字符串里的单引号有点儿多余。

### 7.2.7 占位符与参数绑定

前面在构造查询命令字符串的时候都是把要插入或者要用在筛选条件里的数据值直接放到查询命令字符串里的。这样做有一定的道理，但还有更好的办法。DBI允许我们在构造查询命令字符串的时候先在有关位置放上一些特殊的符号——即所谓的占位符，然后在执行这个查询命令的时候再用具体的数据值替换掉这些占位符。这种做法有两个好处：首先，不需要明确地调用quote()方法就能获得与使用了这个方法同样的效果；其次，如果查询命令会反复执行很多次（比如在一个循环结构里），这将改善脚本的执行性能。

为了把占位符的作用和使用方法讲清楚，我们来看一个例子。假设现在正是一个新学期的开始，需要把考试记分项目中的student数据表里的旧记录清理干净，并用保存在某个文件里的新学生名单来重新建立这个数据表。如果不使用占位符，就得像下面这样去删除member数据表里的旧数据和加载新数据：

```

$dbh->do (qq{ DELETE FROM student } ); # delete existing rows
while (<>)                               # add new rows
{
    chomp;
    $_ = $dbh->quote ($_);
    $dbh->do (qq{ INSERT INTO student SET name = $_ });
}

```

这段代码有两个地方值得改进：首先，它需要由你本人去调用quote()方法来处理数据值里的特殊字符；其次，它的执行效率太低——虽然INSERT查询的基本格式都差不多，可do()方法却必须在每次循环里对prepare()和execute()各做一次调用。要是能做到：在进入循环之前，只在构造INSERT语句的时候调用一次prepare()；在进入循环之后，每次循环只需调用execute()而不必再调用prepare()，即只调用一次prepare()方法，执行效率将大大提高。利用DBI脚本里的占位符，就可以实现这两项改进：

```

$dbh->do (qq{ DELETE FROM student } ); # delete existing rows
my $sth = $dbh->prepare (qq{ INSERT INTO student SET name = ? });
while (<>)                               # add new rows
{
    chomp;
    $sth->execute ($_);
}

```

一般说来,如果发现自己需要在一个循环里反复多次地调用do()方法,就应该考虑把相应的prepare()调用安排在进入循环之前,在循环里则只调用相应的execute()方法。注意到上面那条INSERT查询里的问号“?”字符了吗?它就是占位符。进入循环之后,就在调用execute()方法的时候,占位符将被替换为一个具体的数据值,而脚本代码发送给MySQL服务器的将是一条“新”构造出来的查询命令。同时,DBI将自动给数据值里的特殊字符加上必要的引号,这样就不用不着去调用quote()方法了。

在使用占位符的时候,需要注意以下几个事项:

- 不要给查询命令字符串里的占位符加上引号。如果加上了引号,它将不会被识别为一个占位符。
- 不要使用quote()方法对将来用来替换占位符的数据值进行预处理,否则,这个数据值将会有多余的引号。
- 每个占位符只能对应于一个数值。比如说,不能像下面这样去构造和执行一条语句:

```
my $sth = $dbh->prepare (qq{
    INSERT INTO member last_name, first_name VALUES(?)
});
$sth->execute ("Adams,Bill,2003-09-19");
```

必须像下面这样做:

```
my $sth = $dbh->prepare (qq{
    INSERT INTO member last_name, first_name VALUES(?,?,?)
});
$sth->execute ("Adams","Bill","2003-09-19");
```

- 如果需要把某个占位符替换为NULL值,必须使用undef。
- 占位符和quote()方法只适用于数据值,SELECT之类的关键字或者数据库名、数据表名、数据列名之类的标识符是不允许用占位符来“占位子”的。如果这样做,查询命令字符串里的关键字或标识符就会被加上引号,从而导致这条查询命令因语法错误而执行失败。

对于某些数据库引擎,占位符除了能改善循环语句的执行效率以外,还有另外一个与执行效率有关的好处。这类数据库引擎会把你构造出来的查询命令和它们为这个查询命令而生成的执行计划保存在一个缓存区里。这样,当服务器再次接收到同样的查询请求时,就可以立刻开始执行而不必再去生成一个新的执行计划了。把查询缓存起来的做法特别有利于复杂的SELECT语句,因为好的执行计划通常要多花一些时间才能生成出来。有了占位符,服务器在缓存区里找到一个同样查询的概率就会大大增加——使用了占位符的查询命令肯定要比直接使用具体数据值的查询命令更具有普遍性。不过,MySQL现在还没有这种对查询进行缓存的机制,所以DBI脚本里的占位符现在还不能像前面介绍的那样去改善MySQL的执行效率。可如果把使用了占位符的查询命令移植到某个有这种机制的数据库引擎里,就可以享受到这种性能改善了。(MySQL 4.0.1及以后的版本有一种查询缓存机制,但它缓存的是内容完全相同的查询命令字符串的结果集而不是它们的执行计划。在第4章里曾对MySQL的查询缓存机制进行了讨论。)

### 多余的undef

有些能够用来执行查询命令字符串的DBI方法（比如do()和selectrow\_array()）允许你为查询命令里的任意“?”字符提供替换值。比如说，可以像下面这样去修改一条数据记录：

```
my $rows = $dbh->do ("UPDATE member SET expiration = ? WHERE member_id = ?",
                    undef, "2005-01-01", 14);
```

或者像下面这样去检索一条数据记录：

```
my $ref = $dbh->selectrow_arrayref (
    "SELECT * FROM member WHERE member_id = ?",
    undef, 14);
```

但是，你们是否注意到上面两段代码都有一个“多余”的undef出现在真正的替换值之前呢？造成这种现象的原因是：在这些允许使用占位符作为其输入参数的查询执行类方法里，虽然没有写出但在真正的数据参数之前还存在着一个用来设定一些查询处理属性的参数。这些属性虽然很少用到（如果真有人用的话），但与之对应的输入参数却不能省略——即“多余的”undef。

### 7.2.8 把查询结果绑定给脚本变量

占位符允许你直到执行查询命令时才把真正要用到的数据值替换到查询命令字符串里去。从某种意义上讲，这相当于允许查询命令使用“输入参数”。与此相对应，DBI还提供了一种称为“参数绑定”的输出操作以允许查询命令使用“输出参数”，这个操作在你取回一个数据行时能够自动地把各有关数据列的取值检索到一些变量里去，不需要由你本人去对这些变量进行赋值操作。

假设有一个用来从member数据表检索会员姓名的查询，可以让DBI把所选取的数据列的取值自动赋值给一些Perl变量。当取回一个数据行的时候，这些变量将自动刷新为相应的数据列取值，这就使检索操作非常有效。下面这个例子演示了如何把数据列绑定到某个变量以及如何在用来取回数据行的循环语句里访问这些变量：

```
my ($last_name, $first_name, $suffix);
my $sth = $dbh->prepare (qq{
    SELECT last_name, first_name, suffix
    FROM member ORDER BY last_name, first_name
});
$sth->execute ();
$sth->bind_col (1, \$last_name);
$sth->bind_col (2, \$first_name);
$sth->bind_col (3, \$suffix);
print "$last_name, $first_name, $suffix\n" while $sth->fetch ();
```

bind\_col()调用必须出现在execute()之后、取回数据行之前，它的输入参数有两个：第一个

是某中选数据列的编号,第二个是绑定给该数据列的变量的引用指针。数据列的编号从1开始。

`bind_col()`方法每次只能绑定一个数据列,如果需要把多个数据列绑定到多个变量,可以使用`bind_columns()`方法来一次性地完成这项工作,如下所示:

```
my ($last_name, $first_name, $suffix);
my $sth = $dbh->prepare (qq{
    SELECT last_name, first_name, suffix
    FROM member ORDER BY last_name, first_name
});
$sth->execute ();
$sth->bind_columns (\$last_name, \$first_name, \$suffix);
print "$last_name, $first_name, $suffix\n" while $sth->fetch ();
```

`bind_columns()`调用也必须出现在`execute()`之后、取回数据行之前。

### 7.2.9 设定MySQL服务器连接参数

在DBI脚本里,建立MySQL服务器连接最简单的办法是把所有的连接参数设定为`connect()`方法的输入参数:

```
my $dsn = "DBI:mysql:db_name:host_name";
my $dbh = DBI->connect ($dsn, user_name, password);
```

如果省略了某些连接参数,DBI将根据以下规则去确定之:

- 如果事先设定了`DBI_DSN`环境变量且`connect()`方法里的数据源名 (data source name, DSN) 没有定义或者是空字符串,就将使用`DBI_DSN`环境变量的值作为数据源。如果事先设定了`DBI_USER`和`DBI_PASS`环境变量且`connect()`方法里的用户名和口令没有定义 (注意,不包括它们是空字符串的情况),就将使用`DBI_USER`和`DBI_PASS`环境变量的值作为用户名和口令。在Windows系统上,如果用户名没有定义,就将使用`USER`环境变量的值作为用户名。
- 如果省略了主机名,DBI将尝试连接本地主机。
- 如果把用户名设定为`undef`或一个空字符串,就将使用UNIX登录名进行连接。在Windows系统上,默认用户名是ODBC。
- 如果把口令设定为`undef`或一个空字符串,就不发送口令。

可以在DSN连接参数里设定一些选项,它们只能被追加在该字符串的尾部,并且必须用分号彼此隔开。比如说,可以用`mysql_read_default_file`选项来指定选项文件路径名:

```
my $dsn = "DBI:mysql:sampdb;mysql_read_default_file=/u/paul/.my.cnf";
```

这将使DBI脚本在执行时到指定文件里去读取MySQL服务器的连接参数。比如说,如果`/u/paul/.my.cnf`文件有着如下所示的内容:

```
[client]
host=cobra.snake.net
user=sampadm
password=secret
```

connect()调用就将尝试以用户名 sampadm 和口令 secret 来连接主机 cobra.snake.net 上的 MySQL 服务器。在 UNIX 系统上, 还可以用下面这个办法让脚本去读取当前用户的选项文件:

```
my $dsn = "DBI:mysql:sampdb:mysql_read_default_file=$ENV{HOME}/.my.cnf";
```

\$ENV{HOME} 给出了这个脚本的当前用户的登录主目录 (home directory, 也叫做主目录) 的路径名, 所以 MySQL 服务器的连接参数将来自那位当前用户的选项文件。如果以这种方式来编写 DBI 脚本, 就用不着在脚本代码里写出 MySQL 服务器的连接参数了。

mysql\_read\_default\_file 选项只能让脚本去读取指定选项文件里的连接参数, 如果还想让它去读取全局选项文件 (比如 UNIX 系统上的 /etc/my.cnf 文件或者 Windows 系统上的 C:\my.cnf 文件) 里的连接参数, 这个选项就不能胜任了。如果想让脚本依次读取所有标准选项文件里的连接参数, 就应该使用 mysql\_read\_default\_group 选项。这个选项将把各标准选项文件里的 [client] 设置段和所指定的那个设置段里的连接参数都读取出来。比如说, 如果有一些专供与 sampdb 数据库有关的脚本使用的选项, 就可以把它们列在 [sampdb] 设置段里, 然后在有关的脚本里对 DSN 连接参数做出如下的设置:

```
my $dsn = "DBI:mysql:sampdb:mysql_read_default_group=sampdb";
```

如果只想使用各标准选项文件里的 [client] 设置段里的连接参数, 就应该这样做:

```
my $dsn = "DBI:mysql:sampdb:mysql_read_default_group=client";
```

mysql\_read\_default\_file 和 mysql\_read\_default\_group 这两个选项要求 MySQL 的版本号不得低于 3.22.10, DBD::mysql 的版本号不得低于 1.21.06。关于数据源连接参数选项的详细介绍请参见附录 G, 关于 MySQL 选项文件格式的详细介绍请参见附录 E。

但是, 在 Windows 系统上使用 mysql\_read\_default\_file 选项会遇到这样一个难题: Windows 的文件路径名通常以一个驱动器盘符和一个冒号 (:) 开始, 可 DBI 却会把这个冒号解释为 DSN 字符串里的分隔符。虽说有一个能绕开这一限制的办法, 但这个办法却显得有点笨拙:

1) 先把路径切换到选项文件所在驱动器的根目录, 这样就可以用一个不带驱动器盘符的相对路径来给出选项文件的路径名了。

2) 把 DSN 字符串里的 mysql\_read\_default\_file 选项设置为选项文件的文件名, 但不要写出该驱动器的盘符和紧跟在盘符后面的分号。

3) 如果还想在连接上 MySQL 服务器之后再回到当前目录里, 就必须在调用 connect() 之前先把当前目录的路径名保存起来, 等连接操作完成后, 再通过 chdir() 调用重新回到它里面去。

下面这段代码演示了怎样才能读取选项文件 C:\my.cnf 里的连接参数。(请注意: 在 Perl 字符串里, Windows 路径名里的反斜线 “\” 必须写成斜线 “/”。)

```
# save current directory pathname
use Cwd;
my $orig_dir = cwd ();
# change to root dir of drive where file is located
chdir ("C:/") or die "cannot chdir: $!\n";
# connect using parameters in C:\my.cnf
my $dsn = "DBI:mysql:sampdb:localhost:mysql_read_default_file=/my.cnf";
```



```
my $dbh = DBI->connect ($dsn, undef, undef,
                        { RaiseError => 1, PrintError => 0 });
# change back to original directory
chdir ($orig_dir) or die "cannot chdir: $!\n";
```

在使用了选项文件的情况下,仍可以在connect()调用里设定连接参数(比如说,想让脚本以某个特定的用户身份去执行)。在connect()调用里设定主机名、用户名、口令将优先于脚本在选项文件里找到的连接参数。利用这一点,就能让DBI脚本在连接MySQL服务器时使用我们在命令行上给出的--host和--user选项而不是它在选项文件里找到的选项,这正是各种MySQL客户程序的标准做法。如果没有特殊的理由,在编写DBI脚本时还是沿袭这种标准做法比较好。

在本章后面给出的示例脚本里,将统一使用一些“标准的”代码来完成脚本程序与MySQL服务器的连接和断开工作。为了让大家能够把注意力集中到各有关脚本的重点概念上去,下面给出这些“标准的”代码,以后就不再啰嗦了:

```
#!/usr/bin/perl -w

use DBI;
use strict;

# parse connection parameters from command line if given

use Getopt::Long;
$Getopt::Long::ignorecase = 0; # options are case sensitive
$Getopt::Long::bundling = 1;   # -uname = -u name, not -u -n -a -m -e

# default parameters - all undefined initially
my ($host_name, $password, $port_num, $socket_name, $user_name);

GetOptions (
    # =i means an integer argument is required after option
    # =s or :s means string argument is required or optional after option
    "host|h=s"      => \$host_name,
    "password|p:s"  => \$password,
    "port|P=i"      => \$port_num,
    "socket|S=s"    => \$socket_name,
    "user|u=s"      => \$user_name
) or exit (1);

# solicit password if option specified without option value
if (defined ($password) && !$password)
{
    # turn off echoing but don't interfere with STDIN
    open (TTY, "/dev/tty") or die "Cannot open terminal\n";
    system ("stty -echo < /dev/tty");
    print STDERR "Enter password: ";
    chomp ($password = <TTY>);
    system ("stty echo < /dev/tty");
```

```

    close (TTY);
    print STDERR "\n";
}

# construct data source
my $dsn = "DBI:mysql:sampdb";
$dsn .= ";host=$host_name" if $host_name;
$dsn .= ";port=$port_num" if $port_num;
$dsn .= ";mysql_socket=$socket_name" if $socket_name;
$dsn .= ";mysql_read_default_group=client";

# connect to server
my $dbh = DBI->connect ($dsn, $user_name, $password,
                        { RaiseError => 1, PrintError => 0 });

```

这段代码对DBI进行了初始化。它先对命令行上给出的连接参数（如果有的话）进行了必要的处理，然后使用来自命令行或者它在标准选项文件的[client]设置段里找到的连接参数去连接MySQL服务器。如果把连接参数安排在了选项文件里，就用不着在命令行上给出它们了——当然，脚本里必须有这段“标准的”代码才行。

各有关脚本的结尾部分也都是同样的代码，它们将断开脚本程序与MySQL服务器的连接并退出执行，如下所示：

```

$dbh->disconnect ();
exit (0);

```

在第7.4节讲到DBI脚本的Web编程技术时，我们将对这里给出的连接代码稍微做些修改，但基本思路仍是相同的。

用来提示输入口令的那部分代码只能在UNIX系统上工作。对于Windows系统，建议或者把口令放到某个标准选项文件的[client]设置段里，或者直接在命令行上给出口令。

有一点需要大家特别注意：标准的MySQL客户程序与Getopt模块在处理命令行参数时的做法是有差异的。如果使用了Getopt模块，在执行DBI脚本的时候，口令选项（--password或-p）的后面就必须跟有一个参数值（即口令文本），除非把口令选项放在命令行的最末尾或者在它的后面紧跟一个另外的选项。我们来看一个例子。假设脚本还需要在有关选项的后面再增加一个数据表名作为输入参数。那么，如果像下面这样来调用它，Getopt就会把那个数据表的名字mytbl误认为是口令选项-p的参数值，而不再提示你输入一个口令：

```
% ./myscript.pl -u paul -p mytbl
```

要想让这个脚本正确地提示你输入一个口令，就必须把-p选项挪到-u选项的前面去：

```
% ./myscript.pl -p -u paul mytbl
```

### 7.2.10 调试

如果DBI脚本工作不正常，就需要对它进行调试。有两种比较常用的调试方法，它们既可以单独使用，也可以联合作战。第一种办法是在脚本里安排一些打印语句，它的好处是可以随意

安排调试工作的输出信息,但必须由你本人一条一条地把这些打印语句添加到脚本代码里去。第二种办法是使用DBI内建的跟踪调试功能,它的好处是更普遍和更系统化,在激活后不需要人为干预。DBI内建的跟踪调试功能还能显示数据库驱动程序的操作信息,这是其他的调试手段做不到的。

#### 1. 利用print语句进行调试

在MySQL邮件列表上,经常能看到这样的求助信息:“我的查询命令在mysql客户程序里执行得挺好的,可在DBI脚本里就不行了。为什么会这样?”这类问题最为常见的起因是:由DBI脚本发送出去的查询命令并不是求助者心目中认定的那一个。如果那些求助者把那条由DBI脚本发送到MySQL服务器去的查询命令打印出来的话,就往往会惊讶于他们所看到的东西。我们来看一个例子。假设在mysql客户程序里敲入了下面这个查询命令并得到了正确的执行:

```
mysql> INSERT INTO member (last_name,first_name,expiration)
-> VALUES('Brown','Marcia','2005-6-3');
```

接着,在一个DBI脚本里试图去做同样的事情(记得要去掉SQL语句末尾的分号):

```
$last = "Brown";
$first = "Marcia";
$expiration = "2005-6-3";
$query = qq{
    INSERT INTO member (last_name,first_name,expiration)
    VALUES($last,$first,$expiration)
};
$rows = $dbh->do ($query);
```

查询还是那个查询,但这一次它却没有正确执行。查询真的还是那个查询吗?把它打印出来看看:

```
print "$query\n";
```

这条print语句的输出结果如下所示:

```
INSERT INTO member (last_name,first_name,expiration)
VALUES(Brown,Marcia,2005-6-3)
```

根据这个输出结果,我们可以非常清楚地看出前、后两条查询命令根本就不一样:DBI脚本里的那条查询命令没有给VALUES()列表里的数据值加上引号。这个漏洞有两种办法可以弥补。其一,可以使用前面介绍过的quote()方法,如下所示:

```
$last = $dbh->quote ("Brown");
$first = $dbh->quote ("Marcia");
$expiration = $dbh->quote ("2005-6-3");
$query = qq{
    INSERT INTO member (last_name,first_name,expiration)
    VALUES($last,$first,$expiration)
};
$rows = $dbh->do ($query);
```

其二,可以先使用一些占位符来构造那个查询命令字符串,然后再把将要插入的数据值传

递给do()方法作为其“输入参数”，如下所示：

```
$last = "Brown";
$first = "Marcia";
$expiration = "2005-6-3";
$query = qq{
    INSERT INTO member (last_name,first_name,expiration)
    VALUES(?,?,?)
};
$rows = $dbh->do ($query, undef, $last, $first, $expiration);
```

不过，如果使用的是第二种办法，就无法利用打印语句去查看查询命令的完整内容了，因为用数据值来替换占位符的动作只有在脚本执行到do()方法时才会发生。如果在构造查询命令字符串的时候使用了占位符，DBI内建的跟踪调试机制往往会更有用。

## 2. 利用DBI内建的跟踪调试机制进行调试

DBI提供了一个能够生成各种调试信息的跟踪调试机制，这些信息能帮你找出脚本工作不正常的原因。该机制从0（关闭）到9（信息量最大）共有10个跟踪级别。一般说来，跟踪级别1和2最有用。跟踪级别2能显示正在执行的查询命令的文本（包括占位符的替换结果）、quote()方法的调用结果以及其他一些诸如此类的调试信息，这些信息对调试工作有着巨大的帮助。

可以只对某一个脚本进行跟踪调试（办法是在该脚本里调用trace()方法），也可以对启动运行的全体DBI脚本都进行跟踪调试（办法是设置DBI\_TRACE环境变量）。

在调用trace()方法时，需要给它提供一个跟踪级别参数和一个可选的文件名。如果没有给出这个文件名，调试信息将被发送到系统的标准出错设备STDERR；如果给出了这个文件名，调试信息就将发送到指定的文件里。下面这个调用启动了跟踪级别1，调试信息将被送往STDERR设备：

```
DBI->trace (1);
```

下面这个调用启动了跟踪级别2，调试信息将被送往trace.out文件：

```
DBI->trace (2, "trace.out");
```

如果想关闭跟踪调试功能，请把跟踪级别设定为0：

```
DBI->trace (0);
```

如果被调用为DBI->trace ()，那么将跟踪所有DBI操作。如果不想让牵涉面过大，也可以只跟踪某些具体的DBI句柄。如果基本上可以肯定问题出在脚本的哪几行上，就没有必要激活全局性的跟踪机制——毕竟在一大堆跟踪输出里找到想要的东西也不是件很容易的事，所以只跟踪某些具体的DBI句柄的做法往往更好。比如说，如果能肯定问题就出在某个特定的SELECT语句上，那只需跟踪与该语句相关联的语句句柄就足够了：

```
$sth = $dbh->prepare (qq{ SELECT ... }); # create the statement handle
$sth->trace (1);                          # enable tracing on the statement
$sth->execute ();
```

如果在trace()调用里还给出了一个文件名，那么所有的跟踪输出（不管它们是来自DBI全局

还是来自某个特定的句柄)就都将被写入这个文件。

除进行trace()调用的办法外,还可以使用TraceLevel属性,它最早出现于DBI 1.21版本。这个属性允许设置或者读取某给定句柄上的跟踪级别,如下所示:

```
$dbh->{TraceLevel} = 3;           # set database handle trace level
my $cur_level = $sth->{TraceLevel}; # get statement handle trace level
```

如果想激活全局性的跟踪调试机制,使它能够作用于所运行的全体脚本,可以在shell里设置DBI\_TRACE环境变量。用来设置这个环境变量的语法随shell的不同而有所差异:

- 如果使用的shell是csh或tcsh,则:

```
% setenv DBI_TRACE value
```

- 如果使用的shell是sh、ksh或bash,则:

```
$ export DBI_TRACE=value
```

- 如果使用的是Windows系统,则:

```
C:\> set DBI_TRACE=value
```

value的格式在各种shell里都是一样的:如果它是一个整数 $n$ ,则表示跟踪级别是 $n$ ,跟踪输出将被写到STDERR;如果它是一个文件名,则表示跟踪级别是2,跟踪输出将被写到指定文件;如果它是 $n=file\_name$ ,则表示跟踪级别是 $n$ ,跟踪输出将被送往指定文件。下面是几个使用tcsh语法的例子:

- 跟踪级别是1,跟踪输出将被写到STDERR:

```
% setenv DBI_TRACE 1
```

- 跟踪级别是1,跟踪输出将被写到trace.out文件:

```
% setenv DBI_TRACE 1=trace.out
```

- 跟踪级别是2,跟踪输出将被写到trace.out文件:

```
% setenv DBI_TRACE trace.out
```

使用DBI\_TRACE环境变量的好处是既能激活跟踪调试机制,又不用对脚本做任何修改。但必须提醒大家注意的是,如果从shell里激活了对某个脚本的跟踪调试机制,千万记住要在解决了有关问题之后再把它关闭掉。要知道,调试输出信息将一直追加在跟踪文件的末尾而不会覆盖掉它原有的内容,如果不加注意,这个文件就有可能变得非常巨大。因此,在某个shell启动文件(如.cshrc、.tcshrc、.login或.profile等)里定义DBI\_TRACE环境变量的做法是极不可取的!

- 在csh或tcsh里,下面两条命令都可以把跟踪调试功能关闭掉:

```
% setenv DBI_TRACE 0
% unsetenv DBI_TRACE
```

- 在sh、ksh或bash里,下面的命令可以把跟踪调试功能关闭掉:

```
$ export DBI_TRACE=0
```



- 在Windows系统上, 下面两条命令都可以把跟踪调试功能关闭掉:

```
C:\> unset DBI_TRACE
C:\> set DBI_TRACE=0
```

### 7.2.11 结果集元数据的使用

可以使用DBI来访问结果集元数据 (metadata), 这些元数据是一些关于查询命令所选取的数据行的描述性信息。元数据要通过与结果集相关联的查询语句句柄的属性来访问。它们有的是各种数据库驱动模块都具备的DBI标准属性 (比如NUM\_OF\_FIELDS, 结果集里的数据列个数); 另外一些则是DBD::mysql模块为MySQL提供的专用属性, 这类专用属性 (比如mysql\_max\_length属性, 各数据列的数据值最大宽度) 往往不适用于其他的数据库引擎。如果在脚本里使用了MySQL的专用属性, 则脚本可能无法移植到其他的数据库里去使用。但从另一方面讲, 它们又的确能让你更容易地获得想要的信息。

必须掌握好访问元数据的时机。就拿SELECT语句来说吧, 在完成对它的prepare()和execute()调用之前, 其结果集的属性是不存在的。此外, 当使用某个数据行取回函数到达其结果集的末尾或者调用了finish()方法之后, 这些属性又有可能已经失效了。

下面的示例代码演示了DBI元数据通用属性NUM\_OF\_FIELDS (结果集里的数据列个数) 和NAME (结果集里各数据列的名称) 以及MySQL元数据的专用属性mysql\_max\_length的用法。利用这几个属性提供的信息, 对于来自同一个SELECT查询的结果集, box\_out.pl脚本生成的输出报告将与交互式客户程序mysql生成的输出报告有着同样的表格型格式。以下代码就是box\_out.pl脚本的主要部分 (可以把脚本代码中的SELECT语句任意替换为其他的语句, box\_out.pl脚本中的结果集输出例程与具体的查询命令无关):

```
my $sth = $dbh->prepare (qq{
    SELECT last_name, first_name, suffix, city, state
    FROM president ORDER BY last_name, first_name
});
$sth->execute (); # attributes should be available after this call

# actual maximum widths of column values in result set
my @wid = @{$sth->{mysql_max_length}};
# number of columns in result set
my $ncols = $sth->{NUM_OF_FIELDS};

# adjust column widths if data values are narrower than column headings
# or than the word "NULL"
for (my $i = 0; $i < $ncols; $i++)
{
    my $name_wid = length ($sth->{NAME}->[$i]);
    $wid[$i] = $name_wid if $wid[$i] < $name_wid;
    $wid[$i] = 4 if $wid[$i] < 4;
}
```

```

# print output
print_dashes (\@wid, $ncols);          # row of dashes
print_row ($sth->{NAME}, \@wid, $ncols); # column headings
print_dashes (\@wid, $ncols);          # row of dashes
while (my $ary_ref = $sth->fetchrow_arrayref ())
{
    print_row ($ary_ref, \@wid, $ncols); # row data values
}
print_dashes (\@wid, $ncols);          # row of dashes

```

在调用execute()方法初始化完查询命令之后,我们立刻去访问了所需要的元数据。\$sth->{NUM\_OF\_FIELDS}是一个标量变量,它能告诉我们结果集里有多少个数据列。\$sth->{NAME}和\$sth->{mysql\_max\_length}则分别给出了那些数据列的名字和它们的数据值最大宽度,这两个属性的取值都是一个数组引用指针,数组里的各个元素分别对应着结果集里的各个数据列,它们的排列顺序与有关数据列在查询命令里出现的先后次序相同。

示例代码中的有关计算与第6章里编写的client4程序大同小异。比如说,为避免出现输出报告排列不整齐的现象,如果某数据列名的长度大于其数据值的长度,与之对应的输出列的宽度就将相应地调整为该数据列名的长度。

下面是具体完成结果集打印输出工作的print\_dashes()和print\_row()函数的代码。它们也与第6章里的client4程序中的有关代码大同小异:

```

sub print_dashes
{
    my $wid_ary_ref = shift;    # reference to array of column widths
    my $cols = shift;          # number of columns

    print "+";
    for (my $i = 0; $i < $cols; $i++)
    {
        print "-" x ($wid_ary_ref->[$i]+2) . "+";
    }
    print "\n";
}

# print row of data. (doesn't right-align numeric columns)

sub print_row
{
    my $val_ary_ref = shift;    # reference to array of column values
    my $wid_ary_ref = shift;    # reference to array of column widths
    my $cols = shift;          # number of columns

    print "|";
    for (my $i = 0; $i < $cols; $i++)
    {
        printf " %-*s |", $wid_ary_ref->[$i],

```

```

        defined ($val_ary_ref->[$i]) ? $val_ary_ref->[$i] : "NULL";
    }
    print "\n";
}

```

下面是用box\_out.pl脚本生成的一份输出报告:

```

+-----+-----+-----+-----+-----+
| last_name | first_name | suffix | city | state |
+-----+-----+-----+-----+-----+
| Adams | John | NULL | Braintree | MA |
| Adams | John Quincy | NULL | Braintree | MA |
| Arthur | Chester A. | NULL | Fairfield | VT |
| Buchanan | James | NULL | Mercersburg | PA |
| Bush | George H.W. | NULL | Milton | MA |
| Bush | George W. | NULL | New Haven | CT |
| Carter | James E. | Jr. | Plains | GA |
...

```

下一个脚本将利用数据列的元数据生成另外一种格式的输出报告。这个名为show\_member.pl的脚本能让你快速查看“美国历史研究会”会员的个人资料而无须输入任何查询命令。给定某位会员的姓氏,这个脚本将把中选记录项显示为如下所示的格式:

```

% ./show_member.pl artel
last_name: Artel
first_name: Mike
suffix:
expiration: 2006-04-16
email: mike_artel@venus.org
street: 4264 Lovering Rd.
city: Miami
state: FL
zip: 12777
phone: 075-961-0712
interests: Civil Rights, Education, Revolutionary War
member_id: 63

```

show\_member.pl脚本还接受会员编号或模式匹配模板(用来匹配多个姓氏)作为其输入参数。下面的第一条命令将检索出23号会员的个人资料,第二条命令将检索出那些姓氏以字母“C”开头的会员们的个人资料:

```

% ./show_member.pl 23
% ./show_member.pl C%

```

下面是show\_member.pl脚本主要部分的代码。各输出行的标签是利用NAME属性而确定的,结果集里的数据列个数则是利用NUM\_OF\_FIELDS属性而确定的:

```

my $count = 0; # number of entries printed so far
my @label = (); # column label array
my $label_wid = 0;

```

```

while (@ARGV)          # run query for each argument on command line
{
    my $arg = shift (@ARGV);

    # default is to do a search by last name...
    my $clause = "last_name LIKE " . $dbh->quote ($arg);
    # ...but do ID search instead if argument is numeric
    $clause = "member_id = " . $dbh->quote ($arg) if $arg =~ /\d+$/;

    # issue query
    my $sth = $dbh->prepare (qq{
        SELECT * FROM member
        WHERE $clause
        ORDER BY last_name, first_name
    });
    $sth->execute ();

    # get column names to use for labels and
    # determine max column name width for formatting
    # (only do this the first time through the loop, though)
    if ($label_wid == 0)
    {
        @label = @{$sth->{NAME}};
        foreach my $label (@label)
        {
            $label_wid = length ($label) if $label_wid < length ($label);
        }
    }

    # read and print query results
    my $matches = 0;
    while (my @ary = $sth->fetchrow_array ())
    {
        # print newline before 2nd and subsequent entries
        print "\n" if ++$count > 1;
        foreach (my $i = 0; $i < $sth->{NUM_OF_FIELDS}; $i++)
        {
            # print label
            printf "%-*s", $label_wid+1, $label[$i] . ":";
            # print value, if there is one
            print " " . $ary[$i] if defined ($ary[$i]);
            print "\n";
        }
        ++$matches;
    }
    print "\nNo match was found for \"$arg\"\n" if $matches == 0;
}

```

不管一个数据行包含有多少个数据，show\_member.pl脚本都能把它的完整内容显示出来。这个脚本先使用SELECT \*把所有的数据列全都选上，然后再利用NAME属性把它们的名字查出来。这样，即使member数据表在今后增加或者删除了一些数据列，这个脚本也不需要修改。

如果只想知道某个数据表里到底有哪些数据列而不打算检索其中的数据行，可以使用一个如下所示的查询命令：

```
SELECT * FROM tbl_name WHERE 1 = 0
```

当像平时那样调用完prepare()和execute()方法之后，就可以从“@{\$sth->{NAME}}”里查知各数据列的名字了。但还要提醒大家的是：这种利用一个“空”查询来查知数据列名字的小技巧虽然适用于MySQL，却不见得能移植到别的数据库引擎里去运用。

如果想进一步了解DBI和DBD::mysql模块提供的各种属性，请参阅附录G。那么，是避免使用MySQL专用属性以保持可移植性好呢？还是以牺牲可移植性为代价去使用它们好呢？这些问题还是由你们自己去判断和决定吧。

### 7.2.12 用DBI脚本来实现事务处理机制

用DBI脚本来实现事务处理机制的办法之一是通过有关代码明确地发出SET AUTOCOMMIT、BEGIN、COMMIT和ROLLBACK等语句。（对这几条语句的详细介绍可以在第3章里查到。）下面将要介绍的是另一种办法：利用DBI模块提供的抽象来实现数据库中的事务处理操作。这个抽象由DBI模块中的一组方法和属性构成，它们将自动发出各种与事务处理有关的SQL语句。可以把使用这个抽象而编写出来的DBI脚本移植到其他也支持事务处理的数据库引擎里去使用，而那些通过有关代码明确地发出与事务处理有关的SQL语句的DBI脚本就不见得能移植到其他数据库引擎上去使用了。

要想使用DBI模块提供的事务处理机制，必须满足以下几个条件：

- 所使用的DBD::mysql模块的版本必须是1.2216或更高。
- MySQL服务器必须允许并支持使用InnoDB或BDB数据表。如何查知MySQL服务器是否支持使用InnoDB或BDB数据表的办法可以在第3章里查到。
- 所使用的数据表必须是支持事务处理的类型——就MySQL而言，它们必须是InnoDB或BDB类型的数据表。如果不是这样，就必须先用ALTER TABLE命令改变它们的类型。下面这条语句能把给定数据表tbl\_name的类型修改为InnoDB：

```
ALTER TABLE tbl_name TYPE = INNODB;
```

如果以上条件全部得到了满足，就可以在DBI脚本里按以下步骤实现事务处理机制了：

1) 禁用（或者临时挂起）自动提交模式，这将使SQL语句对数据库的修改只有在发出了commit()调用之后才会永久性地反映在数据库里。

2) 在一个eval语句块里依次发出构成这次事务的查询命令，最后一条命令必须是一个用来提交并结束这次事务处理的commit()调用。注意，要在激活RaiseError并禁用PrintError属性的状态下执行它们，也就是说，只要eval语句块里的查询命令有一条执行出错，eval块就会终止执行且不打印出错信息。



3) 在eval语句块执行终止后,立刻检查其终止状态。如果它是因eval块里的某条查询命令执行出错而终止的,就要立刻调用rollback()方法来取消这次事务并给出相应的出错信息(如果需要的话)。

4) 把自动提交模式和出错处理属性回滚为进入这次事务处理之前的状态。

在第3章里有一个事务处理的示例,当时是用在mysql客户程序里以手动方式依次敲入各有关SQL语句的办法来完成事务的。下面,我们将使用一个DBI脚本来达到同样的目的。这个事务的发生场景是这样的:在把学生们的考试成绩录入到score数据表里之后,发现把两位学生的考试分数给弄颠倒了——学号为8的那位学生的考试分数是13,但被错录为18;学号为9的那位学生的考试分数是18,但被错录为13——现在需要把它们再颠倒过来。下面是准备用来纠正这一错误的两条UPDATE语句:

```
UPDATE score SET score = 13 WHERE event_id = 5 AND student_id = 8;
UPDATE score SET score = 18 WHERE event_id = 5 AND student_id = 9;
```

这两条UPDATE语句确实能把那两条记录里的考试分数修改为正确的值,但它们必须被当做一个整体同时执行成功才有意义。在第3章里,我们在这两条UPDATE语句的前后增加了几条SQL语句,它们实现了事务处理流程中的设置自动提交模式、提交事务、回滚等步骤。在一个使用了DBI事务处理机制的Perl脚本里,考试分数的修改工作将由以下代码来完成:

```
my $orig_re = $dbh->{RaiseError}; # save error-handling attributes
my $orig_pe = $dbh->{PrintError};
my $orig_ac = $dbh->{AutoCommit}; # save auto-commit mode

$dbh->{RaiseError} = 1; # cause errors to raise exceptions
$dbh->{PrintError} = 0; # but suppress error messages
$dbh->{AutoCommit} = 0; # don't commit until we say so

eval
{
    # issue the statements that are part of the transaction
    my $sth = $dbh->prepare (qq{
        UPDATE score SET score = ?
        WHERE event_id = ? AND student_id = ?
    });
    $sth->execute (13, 5, 8);
    $sth->execute (18, 5, 9);
    $dbh->commit(); # commit the transaction
};
if ($?) # did the transaction fail?
{
    print "A transaction error occurred: $@\n";
    # roll back, but use eval to trap rollback failure
    eval { $dbh->rollback (); }
}

$dbh->{AutoCommit} = $orig_ac; # restore auto-commit mode
$dbh->{RaiseError} = $orig_re; # restore error-handling attributes
$dbh->{PrintError} = $orig_pe;
```

第一个eval语句块负责完成这次事务，它的终止状态将被保存在变量\$@里返回：如果两条UPDATE语句的执行都没有出错，就将由commit()调用提交这次事务，变量\$@将为空；否则，eval语句块就会立刻终止执行，变量\$@将包含着出错信息。如果事务失败，示例代码会在打印出变量\$@里的出错信息后调用rollback()方法来取消这次事务。注意，为防止因rollback()调用失败而导致脚本退出执行（别忘了，我们还得把自动提交模式和出错处理属性复原为进入这次事务处理之前的状态），我们把rollback()调用也放到了一个eval块里。

在这一章里，DBI脚本所使用的出错处理模式基本上都是激活RaiseError且禁用PrintError属性的，也就是说，它们已经满足了事务处理机制对出错处理模式的要求。因此，上例中用来保存、设置、复原这些属性的代码就显得有点多余。不过，这些“多余的”代码能够确保脚本可以在任何环境中运行——即使事先不知道脚本运行环境中的出错处理属性的设置情况，脚本也能正确地执行。

### 7.3 DBI脚本实战

在了解了DBI程序设计工作中的基本概念之后，下面进入样板数据库去进行一些实战。第1章曾提出了一些程序设计任务，我们将把其中可以用DBI脚本来解决的问题集中在这一节里解决。

对于考试记分项目，我们想把任意一次给定的考试或测验分数检索出来。

对于“美国历史研究会”项目，我们想实现以下几项任务：

- 按不同格式生成会员名录。为研究会的年会生成一份只包含会员姓名的名单，还想生成一份可供打印的会员名录。
- 查出哪些会员需要在近期续交他们的会费以保留其会员资格，并以电子邮件的形式向他们发出通知。
- 编辑会员记录。（你总得为那些续交了会费的会员修改其会员资格失效日期吧。）
- 查找兴趣相同的会员。
- 把会员名录放到网上。

在这些任务里，有些我们可以运用现有知识编写命令行脚本加以解决，另外一些则要等到学完下一节之后才能用与Web服务器配合使用的DBI脚本来解决。到本章结束的时候，还会有一些目标有待我们去实现，将在第8章里完成那些任务。

#### 7.3.1 美国历史研究会：生成会员名录

我们的目标之一是按不同的格式来生成“美国历史研究会”的会员名录。在各种格式中，最简单的莫过于一份只包含会员姓名的花名册了，我们可以利用这份花名册来生成研究会的年会请柬。这份花名册使用普通文本格式就行——因为它只是年会请柬生成程序的一个组成部分，只要能把有关信息传递到年会请柬生成程序里就足够了。

打印版的会员名录就不应该再使用普通文本格式了，它应该有更好的排版效果。这里选择的是RTF格式（Rich Text Format），这是一种由微软公司开发的排版格式，大多数文字处理软件都支持它——微软的Word自不必说，WordPerfect和AppleWorks等文字处理软件也支持这种格式。不

过,不同的字处理软件对RTF格式的支持程度有高低之分,有些字处理软件只能识别和支持最基本的RTF排版命令。因此,为了确保打印版会员名录能够用在各种字处理软件里,我们将使用基本的RTF子集来生成它——具体地说,我们想让Mac OS X操作系统中的TextEdit程序也能读出这里所生成的RTF文档。

从技术角度讲,生成年会请柬和生成打印版会员名录的具体过程并没有本质上的差异——先用一个查询命令把有关的数据项检索出来,再用一个循环结构来取回各项数据并对它进行排版处理。既然如此,就没必要把它们写成两个脚本。因此,只编写一个脚本(`gen_dir.pl`),但这个脚本必须能生成不同格式的输出。这个脚本需要满足以下要求:

1) 在输出会员数据项之前,先要根据输出格式完成必要的初始化工作:如果将要生成的是年会请柬,那就用不着进行特殊的初始化;如果将要生成的是打印版会员名录,就必须先写出一个RTF文档头。

2) 依次取回结果集里的各个数据项,进行必要的排版处理并打印之。

3) 把数据项全部处理完毕之后,进行必要的扫尾和退出工作:如果生成的是年会请柬,就用不着进行特殊的扫尾处理;如果生成的是打印版会员名录,就必须补足一个RTF文档尾。

因为我们今后还需要利用这个脚本来生成其他格式的输出报告,所以这里使用一个“开关箱”来使它具备一些扩展性。这个开关箱其实是一个复合散列列表,它的每个元素分别对应着一种输出格式:键字是某种输出格式的名称(比如“`banquet`”或“`rtf`”),键值(它们也是散列值)给出了这种格式的输出报告在其生成过程中的各个阶段将要调用的函数(一个用来完成初始化工作的函数、一个用来打印各有关数据项的函数、一个用来完成扫尾工作的函数),如下所示:

```
# switchbox containing formatting functions for each output format
my %switchbox =
(
    "banquet" =>                                # functions for banquet list
    {
        "init"      => undef,                    # no initialization needed
        "entry"     => \&format_banquet_entry,
        "cleanup"   => undef                      # no cleanup needed
    },
    "rtf" =>                                     # functions for RTF format
    {
        "init"      => \&rtf_init,
        "entry"     => \&format_rtf_entry,
        "cleanup"   => \&rtf_cleanup
    }
);
```

这样,只需在运行这个脚本的时候在命令行上给出一个输出格式参数,就能得到指定格式的输出报告了,如下所示:

```
% ./gen_dir.pl banquet
% ./gen_dir.pl rtf
```

有了这个开关箱,我们就能很容易地给这个脚本增加其他的输出格式。具体步骤如下:

- 1) 针对输出报告生成过程的不同阶段编写三个函数（初始化、打印、扫尾）。
- 2) 在开关箱里增加一个新元素，其键字是新输出格式的名称，键值则分别指向第1步里编写的那三个函数。
- 3) 当需要以这种新格式来生成输出报告时，执行gen\_dir.pl脚本并在命令行上给出这种新格式的名称。

gen\_dir.pl脚本将根据命令行上的第一个参数来选择适当的开关箱元素并生成相应格式的输出报告，这部分代码如下所示。如果没有在命令行上给出一个输出格式名称或者给出的是一个无效的输出格式名称，脚本将显示一条出错信息并把可用的输出格式列出来。如果在命令行上给出了一个有效的输出格式名称，\$func\_hashref变量就将指向开关箱里的某个散列值：

```
# make sure one argument was specified on the command line
@ARGV == 1
    or die "Usage: gen_dir format_type\nAllowable formats: "
        . join (" ", sort (keys (%switchbox))) . "\n";

# determine proper switchbox entry from argument on command line;
# if no entry is found, the format type is invalid
my $func_hashref = $switchbox{$ARGV[0]};

defined ($func_hashref)
    or die "Unknown format: $ARGV[0]\nAllowable formats: "
        . join (" ", sort (keys (%switchbox))) . "\n";
```

这段代码将根据散列列表%switch box中的某个键字来确定输出报告的格式。如果给出的是一个有效的输出格式名称，开关箱里的某个散列值就会得到匹配，这个脚本就将使用该散列值指定的函数来生成输出报告。如果给出的是一个无效的输出格式名称，开关箱里的散列值将得不到匹配。这种安排有两个好处：其一，不必把输出格式的名称硬编码在格式选择代码里；其二，如果又往开关箱里添加了新散列值（即给这个脚本增加了新的输出格式），这段代码将自动检测出这一情况而不需要修改。

当在命令行上给出了一个有效的输出格式名称时，上面这段代码将把\$func\_hashref变量设置为相应的散列键值（这个键值本身又是一个指向有关函数的散列值）的引用指针。那些函数将被用来生成指定格式的输出报告。下面这段代码将依次完成调用初始化函数、取回和打印有关数据、调用扫尾函数等输出报告生成步骤：

```
# invoke the initialization function if there is one
&{$func_hashref->{init}} if defined ($func_hashref->{init});

# fetch and print entries if there is an entry formatting function
if (defined ($func_hashref->{entry}))
{
    my $sth = $dbh->prepare (qq{
        SELECT * FROM member ORDER BY last_name, first_name
    });
    $sth->execute ();
```

```

while (my $entry_ref = $sth->fetchrow_hashref ("NAME_lc"))
{
    # pass entry by reference to the formatting function
    &{$func_hashref->{entry}} ($entry_ref);
}

# invoke the cleanup function if there is one
&{$func_hashref->{cleanup}} if defined ($func_hashref->{cleanup});

```

我们在这段代码的数据项取回循环里选用了DBI模块提供的fetchrow\_hashref()方法。选用这个方法的理由是：如果需要该循环取回一个数组，排版函数就必须知道各有关数据列在SQL查询命令里的先后顺序。可是，利用\$sth->{NAME}属性（这个属性给出的正是各有关数据列在SQL查询命令里的先后顺序）不是也可以达到这一目的吗？为什么要多费周折呢？因为fetchrow\_hashref()方法返回的散列值引用指针将使我们能够通过\$entry\_ref->{col\_name}语法来使用各有关数据列的值，要是使用NAME属性的话，就没有这么简单了。fetchrow\_hashref()方法使我们能够方便地生成任意格式的输出生报告，因为我们可以通过它返回的散列值直接对各有关数据列的值进行访问和排版处理。

好了，只要再把各输出格式的有关函数（即开关箱数据项中给出的那三个函数）编写出来，gen\_dir.pl脚本就能交付使用了。

#### 1. 生成普通文本格式的会员名录

根据前面的安排，普通文本格式的会员名录将是年会请柬生成程序的一个组成部分。这种输出格式用不着初始化函数和扫尾函数，只要把它的排版函数（format\_banquet\_entry()）编写出来即可。这个函数需要一个指向member记录项的引用指针作为输入参数并把该记录项里的会员姓名打印出来。这个函数的框架如下所示：

```

sub format_banquet_entry
{
    # print member name here, using first_name, last_name, and suffix
    elements of the hash printed to by the function argument
}

```

这个函数本身并不复杂，但怎样处理好会员姓名中的后缀部分却有点麻烦：“Jr.”或“Sr.”等姓名后缀的前面应该有一个逗号和—个空格，而“II”或“III”之类的姓名后缀的前面却只应该有一个空格。如下所示：

```

Michael Alvis IV
Clarence Elgar, Jr.
Bill Matthews, Sr.
Mark York II

```

字母“I”、“V”、“X”在这里被用做表示辈分的罗马数字，这三个字母的不同组合可以表示出从第1代一直到第39代的辈分。如果辈分超出这个范围，就要用到其他的罗马数字，但这几个字母应该足够了。因此，我们将根据姓名后缀是否与下面这个模式相匹配来决定是否需要加上



一个逗号:

```
/^[IVX]+$/
```

在format\_banquet\_entry()函数里, 需要把人名中的各个部分按适当顺序拼凑在一起, 这部分代码是在生成RTF格式的会员名录时也会用到的。因此, 为避免在format\_rtf\_entry()函数里重复书写同样的代码, 把这部分代码写成一个辅助函数:

```
sub format_name
{
    my $entry_ref = shift;

    my $name = $entry_ref->{first_name} . " " . $entry_ref->{last_name};
    if (defined ($entry_ref->{suffix}))          # there is a name suffix
    {
        # no comma for suffixes of I, II, III, etc.
        $name .= "," unless $entry_ref->{suffix} =~ /^[IVX]+$/;
        $name .= " " . $entry_ref->{suffix}
    }
    return ($name);
}
```

有了这个format\_name()函数, 用来打印会员姓名的format\_banquet\_entry()函数就很简单了:

```
sub format_banquet_entry
{
    printf "%s\n", format_name ($_[0]);
}
```

## 2. 生成排版格式的会员名录

与普通文本格式的会员名录相比, 生成RTF格式会员名录的工作要稍微复杂一些。首先, 我们将不得不为每条记录打印出更多的信息; 其次, 为得到我们想要的效果, 既要给每条记录加上一些RTF排版命令, 还必须在文档的开头和结尾加上一些排版命令。RTF文档的最小框架如下所示:

```
{\rtf0
{\fonttbl {\f0 Times;}}
\plain \f0 \fs24
    ...document content goes here...
}
```

文档以左花括号“{”开始, 以右花括号“}”结束。RTF关键字以一个反斜线字符(\)打头, 文档中的第一个关键字必须是“\rtfn”, 其中 $n$ 是该文档所遵守的RTF标准的版本号。根据前面的讨论, 我们将使用“\rtf0”来作为文档的第一个关键字。

在文档内, 我们还需要给出一个字体表来表明文档内容所使用的字体。字体表由一组列在一对花括号中间、以关键字“\fonttbl”打头的字体信息构成。在上面给出的最小框架里, 字体表把0号字体定义为Times字体。(这里只需要一种字体。如果想把文档弄得更花哨一点, 可以再增加几种。)

接下来的几条指令设置了默认的排版风格：“\plain”选择普通格式，“\f0”选择0号字体（已经在字体表里把它定义为Times字体了），“\fs24”把字体大小设置为12点（关键字“\fs”后面的数字以半个点为计量单位）。不需要设置页边距，因为大多数字处理软件都能提供一组很合理的默认值。

打印版会员名录的RTF框架由它的初始化函数和扫尾函数负责提供，下面是这两个函数的代码（为了在输出报告里打印出反斜线字符“\”，需要在脚本代码里把它们双写为“\\”）：

```
sub rtf_init
{
    print "{\\rtf0\n";
    print "{\\fonttbl {\\f0 Times;}}\n";
    print "\\plain \\f0 \\fs24\n";
}

sub rtf_cleanup
{
    print "}\n";
}
```

RTF格式的会员名录的内容将由它的排版函数负责生成。为简单起见，我们将把记录项打印在不同的行上，并给每一行加上一个标签。如果对应于某个输出行的内容缺失，就不打印这一行。（比如说，如果某位会员没有电子邮件地址，就不需要打印“Email:”行。）因为有些输出行（比如“Address:”行）需要由多个数据列（street、city、state、zip）里的信息拼凑出来，所以这个脚本必须能够应付各种数据值缺失的情况。下面是我们使用的输出格式的一个样本：

```
Name: Mike Artel
Address: 4264 Lovering Rd., Miami, FL 12777
Telephone: 075-961-0712
Email: mike_artel@venus.org
Interests: Civil Rights, Education, Revolutionary War
```

这个记录项的RTF表示形式如下所示：

```
\b Name: Mike Artel\b0\par
Address: 4264 Lovering Rd., Miami, FL 12777\par
Telephone: 075-961-0712\par
Email: mike_artel@venus.org\par
Interests: Civil Rights, Education, Revolutionary War\par
```

“Name:”行的黑体字显示效果是靠它首尾两端的“\b”（开始以黑体字显示，后面要有一个空格）和“\b0”（结束黑体字显示）得到的；会员姓名是用上一小节里编写的format\_name()函数拼凑并排版出来的。我们在各行的末尾加上了一个RTF段落标记（“\par”），这个标记告诉字处理软件移动到下一行——没有上面太复杂的东西。这部分代码的难点集中在地址字符串的排版处理和判断哪些输出行需要省略这两方面：

```

sub format_rtf_entry
{
my $entry_ref = shift;

printf "\\b Name: %s\\b0\\par\\n", format_name ($entry_ref);
my $address = "";
$address .= $entry_ref->{street}
            if defined ($entry_ref->{street});
$address .= ", " . $entry_ref->{city}
            if defined ($entry_ref->{city});
$address .= ", " . $entry_ref->{state}
            if defined ($entry_ref->{state});
$address .= " " . $entry_ref->{zip}
            if defined ($entry_ref->{zip});
print "Address: $address\\par\\n"
            if $address ne "";
print "Telephone: $entry_ref->{phone}\\par\\n"
            if defined ($entry_ref->{phone});
print "Email: $entry_ref->{email}\\par\\n"
            if defined ($entry_ref->{email});
print "Interests: $entry_ref->{interests}\\par\\n"
            if defined ($entry_ref->{interests});
print "\\par\\n";
}

```

大家不必拘泥于这里给出的排版样式。只需修改format\_rtf\_entry()函数,就可以任意改变任何一个字段的打印格式,也就是说,可以随心所欲地改变会员名录的排版风格,而这对会员名录的原始格式(即一个字处理文档)来说是相当困难的。

到这里,gen\_dir.pl脚本就编写完成了。现在,只需发出下面这样的命令,就可以生成普通文本或RTF排版格式的会员名录了:

```

% ./gen_dir.pl banquet > names.txt
% ./gen_dir.pl rtf > directory.rtf

```

现在,只需再有一个简单的步骤,就能够把会员名录读出并粘贴到年会请柬程序文档或者把RTF文件读到支持RTF的字处理软件里去了。

DBI使从MySQL数据库提取信息的工作变得简单易行,Perl语言的文本处理能力又使这些信息的排版输出工作变得简单易行。虽然MySQL本身并不具备花哨的排版输出功能,但这并没有多大的关系,因为你们现在已经知道怎样才能把MySQL强大的数据库处理能力与Perl语言出色的文本处理能力集成在一起了。

### 7.3.2 美国历史研究会:发出会费催交通知

在使用原始格式(即一个字处理文档)的“美国历史研究会”会员名录的情况下,想查出需要通知哪些会员续交会费的工作将既耗费时间,又容易出错。现在,当我们把这些信息搬到数据库里以后,发出会费催交通知的工作就可以自动化一些了。我们的目标是先查出有哪些会

员需要续交会费,再通过电子邮件向他们发出会费催交通知。这样,就不必通过电话或者邮政信件来通知他们了。

首先,需要查出有哪些会员应该在指定日期内交纳他们的会费。这个查询将涉及到日期的计算问题,但计算工作相当简单。如下所示:

```
SELECT ... FROM member
WHERE expiration < DATE_ADD(CURDATE(), INTERVAL cutoff DAY)
```

其中, *cutoff* 是我们预定的会费交纳宽限天数。这个查询将把那些应该在这个宽限期内交纳会费的会员记录项检索出来。如果想知道有哪些会员实际上已经失去了会员资格,只需把这个查询命令中的宽限期设置为0就能把失效日期在今天之前的那些记录项给查出来。

在检索出需要发出会费催交通知的记录项之后,下一步该怎么办呢?一种方案是从同一个脚本直接发出电子邮件,但在发出通知之前还是先看看这份名单比较稳妥。因此,我们将分两步来完成这一任务:

1) 通过 *need\_renewal.pl* 脚本查出有哪些会员应该续交会费。应该仔细核对一下这个脚本生成的名单,然后再以它为输入进行第2步工作——发出会费催交通知。

2) 通过 *renewal\_notify.pl* 脚本发出催交会费的电子邮件。如果某位会员没有电子邮件地址,这个脚本将通知你,好让你能够采用其他手段去联络这位会员。

作为发出会费催交通知任务的第1步工作, *need\_renewal.pl* 脚本必须把需要续交会费的会员都查出来。这个脚本用来完成这一查询的代码如下所示:

```
# Use default cutoff of 30 days...
my $cutoff = 30;
# ...but reset if a numeric argument is given on the command line
$cutoff = shift (@ARGV) if @ARGV && $ARGV[0] =~ /\d+$/;

warn "Using cutoff of $cutoff days\n";

my $sth = $dbh->prepare (qq{
    SELECT
        member_id, email, last_name, first_name, expiration,
        TO_DAYS(expiration) - TO_DAYS(CURDATE()) AS days
    FROM member
    WHERE expiration < DATE_ADD(CURDATE(), INTERVAL ? DAY)
    ORDER BY expiration, last_name, first_name
});
$sth->execute ($cutoff);    # pass cutoff as placeholder value

while (my $entry_ref = $sth->fetchrow_hashref ())
{
    # convert undef values to empty strings for printing
    foreach my $key (keys (%{$entry_ref}))
    {
        $entry_ref->{$key} = "" if !defined ($entry_ref->{$key});
    }
}
```

```

print join ("\t",
           $entry_ref->{member_id},
           $entry_ref->{email},
           $entry_ref->{last_name},
           $entry_ref->{first_name},
           $entry_ref->{expiration},
           $entry_ref->{days} . " days")
. "\n";
}

```

need\_renewal.pl脚本的输出如下所示（你们看到的结果应该与此有所不同，因为这个结果是以当前日期为依据的——你们试用这个脚本时的当前日期与这里的当前日期肯定是不一样的）：

```

89  g.steve@pluto.com      Garner  Steve  2002-08-03  -32 days
18  york_mark@earth.com   York    Mark   2002-08-24  -11 days
82  john_edwards@venus.org Edwards John  2002-09-12   8 days

```

请注意，有些会员的会费交纳宽限天数是一个负数，这意味着他们已经丧失了会员资格！（这是因为以前是以手工方式来进行这项工作的，这些会员从你眼皮底下漏了过去。现在，既然我们已经把这些信息搬到了数据库里，那些“漏网之鱼”就无处藏身了。）

作为发出会费催交通知任务的第2步工作，renewal\_notify.pl脚本将以电子邮件的方式发出会费催交通知。为了使renewal\_notify.pl脚本更容易使用，我们可以让它接受三种命令行参数：会员的ID编号、电子邮件地址和文件名。数值参数将被视为会员的ID编号，包含一个“@”字符串的参数将被视为电子邮件地址。其他任何东西都将被视为文件名，renewal\_notify.pl脚本将从这个文件里读出ID编号或电子邮件地址。这样，既可以通过ID编号也可以通过电子邮件地址来指定需要发出会费催交通知的会员，而且，既可以在命令行上直接指定，也可以把他们列在一个文件里。（具体地说，可以把need\_renewal.pl脚本的输出保存为一个文件，再把这个文件用做renewal\_notify.pl脚本的输入。）

对于每一位需要发出会费催交通知的会员，renewal\_notify.pl脚本将从相关的member数据表里的记录项里把其电子邮件地址提取出来，然后向这个地址发出一封电子邮件。如果那个记录项里没有电子邮件地址，renewal\_notify.pl脚本将生成一条警告信息，好让你通过其他手段去与这位会员进行联系。

下面是renewal\_notify.pl脚本用来处理输入参数的主循环。如果没有在命令行上给出任何参数，这个脚本将从标准输入设备上读取输入。如果在命令行上给出了一些参数，它们将被传递到interpret\_argument()函数并在那里被归类为ID编号、电子邮件地址或者文件名：

```

if (@ARGV == 0)      # no arguments, read STDIN for values
{
    read_file (\*STDIN);
}
else
{
    while (my $arg = shift (@ARGV))
    {

```



```

        # interpret argument, with filename recursion
        interpret_argument ($arg, 1);
    }
}

```

read\_file()函数负责读入一个文件（该文件应该已经被打开）的内容并查看每一行的第一个字段（如果我们把need\_renewal.pl脚本的输出馈入renewal\_notify.pl脚本，每一行将有多多个字段，但我们只查看第一个字段，它应该是一个ID编号），如下所示：

```

sub read_file
{
    my $fh = shift;      # handle to open file
    my $arg;

    while (defined ($arg = <$fh>))
    {
        # strip off everything past column 1, including newline
        $arg =~ s/\s.*//s;
        # interpret argument, without filename recursion
        interpret_argument ($arg, 0);
    }
}

```

interpret\_argument()函数用来对每一个参数进行分类以判断它到底是一个ID编号、一个电子邮件地址还是一个文件名。如果是ID编号或电子邮件地址，它将查出相应的会员记录并且把它传递到notify\_member()函数去。要特别注意那些以电子邮件地址指定的会员——有可能出现两位会员使用着同一个电子邮件地址的情况（比如一对夫妻），我们可不想把会费催交通知发送给一个不相干的人。为避免出现这种错误，就得根据电子邮件地址去检查与之对应的ID编号是否只有一个。如果某个电子邮件地址匹配到多个ID编号，因为无法据此断定到底哪位会员没有交纳会费，所以只能在打印出一条警告信息之后忽略它。

如果某个输入参数既不是一个数字也不是一个电子邮件地址，我们将把它视为一个文件名并继续从该文件读取输入数据。这里也必须谨慎从事——为避免陷入无限循环，如果我们已经在读取某个文件的内容，就不应该再开始读取另一个文件。下面是interpret\_argument()函数的代码：

```

sub interpret_argument
{
    my ($arg, $recurse) = @_;

    if ($arg =~ /\d+$/)      # numeric membership ID
    {
        notify_member ($arg);
    }
    elsif ($arg =~ /\@/)      # email address
    {
        # get member_id associated with address
        # (there should be exactly one)
    }
}

```

```

my $query = qq{ SELECT member_id FROM member WHERE email = ? };
my $ary_ref = $dbh->selectcol_arrayref ($query, undef, $arg);
if (scalar (@{$ary_ref}) == 0)
{
    warn "Email address $arg matches no entry: ignored\n";
}
elsif (scalar (@{$ary_ref}) > 1)
{
    warn "Email address $arg matches multiple entries: ignored\n";
}
else
{
    notify_member ($ary_ref->[0]);
}
}
else
    # filename
{
    if (!$recurse)
    {
        warn "filename $arg inside file: ignored\n";
    }
    else
    {
        open (IN, $arg) or die "Cannot open $arg: $!\n";
        read_file (\*IN);
        close (IN);
    }
}
}
}

```

notify\_member()函数负责实际完成发出会费催交通知的工作。如果某位会员没有电子邮件地址，notify\_member()函数就无法发出任何消息，但它会打印出一条警告信息以提醒你需要另想办法去与这位会员进行联系。(利用警告信息中给出的会员ID编号去执行show\_member.pl脚本，就能查看到这位会员的完整资料并找出这位会员的电话号码或地址。)下面是notify\_member()函数的代码：

```

sub notify_member
{
    my $member_id = shift;

    warn "Notifying $member_id...\n";
    my $query = qq{ SELECT * FROM member WHERE member_id = ? };
    my $sth = $dbh->prepare ($query);
    $sth->execute ($member_id);
    my @col_name = @{$sth->{NAME}};
    my $entry_ref = $sth->fetchrow_hashref ();
    $sth->finish ();
}

```

```

if (!$entry_ref)                                # no member found!
{
    warn "NO ENTRY found for member $member_id!\n";
    return;
}
if (!defined ($entry_ref->{email}))              # no email address in entry
{
    warn "Member $member_id has no email address; no message was sent\n";
    return;
}
open (OUT, "| $sendmail") or die "Cannot open mailer\n";
print OUT <<EOF;
To: $entry_ref->{email}
Subject: Your USHL membership is in need of renewal

Greetings.  Your membership in the U.S. Historical League is
due to expire soon.  We hope that you'll take a few minutes to
contact the League office to renew your membership.  The
contents of your member entry are shown below.  Please note
particularly the expiration date.

Thank you.

EOF
foreach my $col_name (@col_name)
{
    printf OUT "$col_name:";
    printf OUT " $entry_ref->{$col_name}"
        if defined ($entry_ref->{$col_name});
    printf OUT "\n";
}
close (OUT);
}

```

notify\_member()函数发出电子邮件的办法是：打开一个通往sendmail程序的管道并把邮件消息的内容馈入这个管道。sendmail程序的路径名在renewal\_notify.pl脚本的开头被设置为一个参数。sendmail程序的位置随系统的不同而不同，所以可能需要对这个路径做相应的修改：

```

# change path to match your system
my $sendmail = "/usr/sbin/sendmail -t -oi";

```

如果系统上没有sendmail程序，这个脚本就无法正确工作。（比如说，Windows系统往往不会安装sendmail程序。）为应对这种情况，sampdb发行版本还收录了renewal\_notify.pl脚本的一个替代版本，它是用Mail::Sendmail模块编写出来的，可以在没有安装sendmail程序的系统上运行。只要系统上安装有Mail::Sendmail模块，就可以使用那个替代版本。

还可以在renewal\_notify.pl脚本的基础上做进一步的发挥——比如说，可以在member数据表里增加一个数据列来记录最近一次会费催交通知是在什么时候发出的，然后让renewal\_notify.pl

脚本在它发出会费催交通知的同时去刷新那个数据列。这有助于使你不至于过于频繁地发出会费催交通知。就目前的现状而言，假设运行这个脚本的频率不会超过每月一次。

这两个脚本到现在就全部完成了。可以按以下步骤来使用它们：首先，运行need\_renewal.pl脚本以生成一份其会员资格已经失效或将在近期失效的会员名单：

```
% ./need_renewal.pl > tmp
```

然后，核对这份名单，看它有没有不合理的地方。如果核对无误，就把它用做renewal\_notify.pl脚本的输入以发出会费催交通知：

```
% ./renewal_notify.pl tmp
```

如果不想以上面这种批处理方式发出会费催交通知，也可以利用ID编号或电子邮件地址来通知有关的会员。比如说，下面的命令将向两位会员（ID编号为18的会员和电子邮件地址是g.steve@pluto.com的会员）分别发出一份会费催交通知：

```
% ./renewal_notify.pl 18 g.steve@pluto.com
```

### 7.3.3 美国历史研究会：编辑会员记录项

在我们发出会费催交通知之后，那些收到通知的人中肯定会有续交会费的。如果有人续交了会费，就需要修改他的会员资格失效日期。在下一章里，我们将把会员记录的编辑工作放到Web上去。但在这一小节里，还是先编写一个命令行脚本（edit\_member.pl）来解决这个问题。这个脚本将使用一个简单的办法（即提示你去输入会员记录各有关数据列的新值）来完成修改会员记录的任务。它的执行情况如下所示：

- 如果在调用edit\_member.pl脚本的时候没有在命令行上给出任何参数，它将假设你是想创建一个新会员。于是，它将提示你输入这个新会员的个人资料，并用你提供的会员资料创建出一个新的记录项来。
- 如果在调用edit\_member.pl脚本的时候在命令行上给出了一个会员ID编号，它将假设你是想修改该会员的个人资料，它将把该会员的记录项检索出来并提示你修改各数据列的值。如果你在某个数据列里输入了一个新值，它将替换掉该数据列的当前值。如果你直接按下回车键，该数据列里的值将不发生变化。如果你输入了单词“none”，它将清除该数据列里的当前值。（如果不知道某位会员的ID编号，可以执行show\_member.pl last\_name命令来查出与给定姓氏last\_name相匹配的记录项里的ID编号值。）

如果想做的仅仅是要修改某位会员的会员资格失效日期的话，专门编写一个能对整个记录进行修改的脚本就未免有些小题大做。但从另一方面讲，如果想让一位不懂SQL的人也能够对记录项里的各个数据列进行各种修改，这样的脚本将提供一种简单而通用的解决方案。（作为一个特例，edit\_member.pl脚本不允许你去修改member\_id字段，因为这个字段里的值将由这个脚本在创建一条新记录时自动生成，一旦生成，就不允许再被改变了。）

edit\_member.pl脚本要做的第一件事是把member数据表里的各数据列名搞清楚：

```

# get member table column names
my $sth = $dbh->prepare (qq{ SELECT * FROM member WHERE 0 });
$sth->execute ();
my @col_name = @{$sth->{NAME}};
$sth->finish ();
Then we can enter the main loop:
if (@ARGV == 0) # if no arguments were given, create a new entry
{
    # pass reference to array of column names
    new_member (\@col_name);
}
else # otherwise edit entries using arguments as member IDs
{
    # save @ARGV, then empty it so that when the script reads from
    # STDIN, it doesn't interpret @ARGV contents as input filenames
    my @id = @ARGV;
    @ARGV = ();
    # for each ID value, look up the entry, then edit it
    while (my $id = shift (@id))
    {
        $sth = $dbh->prepare (qq{
            SELECT * FROM member WHERE member_id = ?
        });
        $sth->execute ($id);
        my $entry_ref = $sth->fetchrow_hashref ();
        $sth->finish ();
        if (!$entry_ref)
        {
            warn "No member with member ID = $id\n";
            next;
        }
        # pass reference to array of column names and reference to entry
        edit_member (\@col_name, $entry_ref);
    }
}
}

```

下面是用来创建一条新会员记录项的有关代码。它先提示你输入member数据表各数据列的值, 然后发出一条INSERT语句来插入一条新记录:

```

sub new_member
{
    my $col_name_ref = shift; # reference to array of column names
    my $entry_ref = { }; # create new entry as a hash

    return unless prompt ("Create new entry (y/n)? ") =~ /^y/i;
    # prompt for new values; user types in new value, or Enter
    # to leave value unchanged, "none" to clear the value, or
    # "exit" to exit without creating the record.

```



```

foreach my $col_name (@{$col_name_ref})
{
    next if $col_name eq "member_id"; # skip key field
    my $col_val = col_prompt ($col_name, undef);
    next if $col_val eq ""; # user pressed Enter
    return if $col_val eq lc ("exit"); # early exit
    $col_val = undef if $col_val eq lc ("none");
    $entry_ref->{$col_name} = $col_val;
}
# show values, ask for confirmation before inserting
show_member ($col_name_ref, $entry_ref);
return unless prompt ("\nInsert this entry (y/n)? ") =~ /^y/i;

# construct an INSERT query, then issue it.
my $query = "INSERT INTO member";
my $delim = " SET "; # put "SET" before first column, "," before others
foreach my $col_name (@{$col_name_ref})
{
    # only specify values for columns that were given one
    next if !defined ($entry_ref->{$col_name});
    # quote() quotes undef as the word NULL (without quotes),
    # which is what we want. Columns that are NOT NULL will
    # be assigned their default values.
    $query .= sprintf ("%s %s=%s", $delim, $col_name,
                        $dbh->quote ($entry_ref->{$col_name}));
    $delim = ",";
}
$dbh->do ($query) or warn "Warning: new entry not created?\n"
}

```

edit\_member.pl脚本使用了两个例程来提示你输入有关信息。prompt()函数用来提出一个问题并返回其答案:

```

sub prompt
{
    my $str = shift;

    print STDERR $str;
    chomp ($str = <STDIN>);
    return ($str);
}

```

col\_prompt()函数需要使用一个数据列名作为其输入参数。它先打印出这个数据列名作为提示, 再把你为这个数据列输入的新值作为自己的返回值:

```

sub col_prompt
{
    my ($col_name, $entry_ref) = @_;

    my $prompt = $col_name;

```

```

if (defined ($entry_ref))
{
    my $cur_val = $entry_ref->{$col_name};
    $cur_val = "NULL" if !defined ($cur_val);
    $prompt .= " [$cur_val]";
}
$prompt .= ": ";
print STDERR $prompt;
my $str = <STDIN>;
chomp ($str);
return ($str);
}

```

col\_prompt()函数的第二个参数是一个散列列表引用指针, 这个散列列表对应着给定会员的记录项。如果是创建新记录项, 这个引用指针的值将是undef; 如果是编辑现有的记录项, 它将指向该记录项的当前内容。在后一种情况里, col\_prompt()函数将把各数据列的当前值也包含在提示字符串里显示出来。也就是说, 在编辑一条现有记录项的时候, 你将能够看到它的各数据列的当前值——如果接受当前值而不需要对之进行修改, 直接按下回车键即可。

用来编辑一个现有会员的代码与用来创建一个新会员的代码很相似。但因为有关的记录项已经存在, 所以提示例程还需要把该记录项的当前值也显示出来, 此外, edit\_member()函数将发出一条UPDATE语句而不是一条INSERT语句:

```

sub edit_member
{
    # references to array of column names and to entry hash
    my ($col_name_ref, $entry_ref) = @_;

    # show initial values, ask for okay to go ahead and edit
    show_member ($col_name_ref, $entry_ref);
    return unless prompt ("\nEdit this entry (y/n)? ") =~ /^y/i;
    # prompt for new values; user types in new value, or Enter
    # to leave value unchanged, "none" to clear the value, or
    # "exit" to exit without changing the record.
    foreach my $col_name (@{$col_name_ref})
    {
        next if $col_name eq "member_id"; # skip key field
        my $col_val = col_prompt ($col_name, $entry_ref);
        next if $col_val eq ""; # user pressed Enter
        return if $col_val eq lc ("exit"); # early exit
        $col_val = undef if $col_val eq lc ("none");
        $entry_ref->{$col_name} = $col_val;
    }
    # show new values, ask for confirmation before updating
    show_member ($col_name_ref, $entry_ref);
    return unless prompt ("\nUpdate this entry (y/n)? ") =~ /^y/i;

    # construct an UPDATE query, then issue it.

```

```

my $query = "UPDATE member";
my $delim = " SET "; # put "SET" before first column, "," before others
foreach my $col_name (@{$col_name_ref})
{
    next if $col_name eq "member_id"; # skip key field
    # quote() quotes undef as the word NULL (without quotes),
    # which is what we want. Columns that are NOT NULL will
    # be assigned their default values.
    $query .= sprintf ("%s %s=%s", $delim, $col_name,
                        $dbh->quote ($entry_ref->{$col_name}));
    $delim = ",";
}
$query .= " WHERE member_id = " . $dbh->quote ($entry_ref->{member_id});
$dbh->do ($query) or warn "Warning: entry not undated?\n"
}

```

edit\_member.pl脚本的一个不足之处是它没有对输入值进行合法性检查。member数据表的大多数字段并不需要进行这种检查——它们都是一些字符串字段。但这个数据表里的会员资格失效日期字段却需要进行这种检查——必须保证这个输入值是一个日期才行。在编写一个通用数据录入程序的时候，应该设法把有关数据表各数据列的类型信息提取出来，然后以此为依据对各有关输入值的合法性进行检查。输入值的合法性检查是一个很复杂的问题，这里不做细致深入的讨论。但在col\_prompt()函数里增加一些代码以检查expiration数据列的输入值格式是否合法。下面就是用来完成对日期值进行最基本的输入检查的代码：

```

sub col_prompt
{
    my ($col_name, $entry_ref) = @_;

loop:
    my $prompt = $col_name;
    if (defined ($entry_ref))
    {
        my $cur_val = $entry_ref->{$col_name};
        $cur_val = "NULL" if !defined ($cur_val);
        $prompt .= " [$cur_val]";
    }
    $prompt .= ": ";
    print STDERR $prompt;
    my $str = <STDIN>;
    chomp ($str);
    # perform rudimentary check on the expiration date
    if ($str && $col_name eq "expiration") # check expiration date format
    {
        if ($str !~ /\d+\D\d+\D\d+$/ )
        {
            warn "$str is not a legal date, try again\n";
            goto loop;
        }
    }
}

```

```

    }
}
return ($str);
}

```

这段代码中的匹配模式能够把三种以非数字字符分隔的数字序列识别为合法的日期值。但这个检查并不完备，因为它会把"1999-14-92"之类的序列也判定为合法的输入值。如果想让这个脚本更加完善，就需要让它对输入值做更严格的检查，甚至需要再增加一些其他的检查，比如要求first\_name（人名）或last\_name（姓氏）字段不得为空等等。

还可以对这个脚本做出另一项改进：如果没有对某条现有记录项进行任何修改，就不发出UPDATE语句。这项改进可以这样来实现：先把各数据列的当前值保存起来，然后只把修改过的数据列放在将由这个脚本提交给MySQL服务器去执行的UPDATE语句里；如果什么都没有改，就根本不发出这条UPDATE语句。另一个值得改进的地方是：如果某条现有记录项在修改期间被别人抢先修改了，这个脚本将通知你。这项改进可以这样来实现：在WHERE子句里为各数据列的原始值分别加上一个AND col\_name = col\_value表达式。这样，一旦有人抢在前面修改了这条记录项，所发出的UPDATE语句就会执行失败，而你也就能够知道还有另外一个人正在试图修改这个记录项了。

edit\_member.pl脚本还有其他一些值得改进的地方。比如说，它会在进入提示循环之前打开一个到数据库的连接，但要一直等到离开提示循环并发出UPDATE语句之后才会关上这个连接。换句话说，如果在输入或修改记录项时花费了很长的时间——或者只是因为临时有事而离开了一会儿，那么在这段时间内，这个连接将一直处于打开状态。这既不安全，也没有效率。那么，怎样修改edit\_member.pl脚本才能让它与数据库的连接保持时间最短（即只在必要时才打开到数据库的连接并在完成有关操作后及时关闭这个连接）呢？这个问题就留给读者自己去解决。

#### 7.3.4 美国历史研究会：查找兴趣相同的会员

人们之所以会加入“美国历史研究会”，一个很重要的原因是希望能在这里找到对美国历史事件（比如大萧条时期或亚伯拉罕·林肯总统的生平等）有着共同兴趣的朋友，而为会员提供一份这样的“好友名单”也正是你（别忘了，你是这个研究会的秘书）的职责之一。当你还使用着被保存为字处理文档格式的会员名录时，因为有字处理软件提供的“查找”功能可用，所以查找这些会员的工作还不算困难。可要想生成一份只包括这些会员在内的名单就不那么容易了，需要大量的复制和粘贴操作。而有了MySQL数据库之后，这项工作就变得简单多了，只需执行一个下面这样的查询就能达到目的：

```

SELECT * FROM member WHERE interests LIKE '%lincoln%'
ORDER BY last_name, first_name

```

不过，如果是在mysql客户程序里执行这个查询的话，其结果看上去并不是非常好。下面，用DBI脚本interests.pl来生成一份更美观的输出报告。这个脚本先要检查我们在命令行上至少给出了一个参数，如果一个参数都没有的话，就没什么可查找的了。然后，对于每个参数，这个脚本在member数据表的interests数据列上运行一个查询，如下所示：

```
@ARGV or die "Usage: interests.pl keyword\n";
search_members (shift (@ARGV)) while @ARGV;
```

在对关键字字符串进行搜索的时候，我们先在它的两端分别加上一个“%”通配符，然后去进行模式匹配。这样，不管这个字符串会出现在interests数据列里的什么位置，我们都能把它给找出来。然后，把匹配到的记录项打印出来：

```
sub search_members
{
    my $interest = shift;

    print "Search results for keyword: $interest\n\n";
    my $sth = $dbh->prepare (qq{
        SELECT * FROM member WHERE interests LIKE ?
        ORDER BY last_name, first_name
    });
    # look for string anywhere in interest field
    $sth->execute ("% " . $interest . "%");
    my $count = 0;
    while (my $hash_ref = $sth->fetchrow_hashref ())
    {
        format_entry ($hash_ref);
        ++$count;
    }
    print "Number of matching entries: $count\n\n";
}
```

format\_entry()函数负责把一个记录项转换为它的可打印形式。这个函数与gen-dir.pl脚本里的format\_rtf\_entry()函数基本一致，只是去掉了后者中的RTF排版命令部分而已，所以它的代码就不在这里重复给出了。如果想了解它的具体实现情况的话，请自行研读sampdb发行版本里的interests.pl脚本。

### 7.3.5 美国历史研究会：把会员名录放到网上

在第7.4节里，我们将开始编写通过显示在客户端Web浏览器里的Web页面去连接MySQL服务器以提取信息和显示查询结果的脚本。那些脚本将根据客户请求去动态地生成各种HTML文档。作为其铺垫，也为了让大家对HTML有一个基本了解，这里先编写一个能够生成一份静态HTML文档的DBI脚本，如果把它生成的HTML文档加载到某个Web服务器的文档树里，就可以通过Web浏览器来查看其内容了。我们现在的任务是生成一份HTML格式的“美国历史研究会”会员名录（别忘了，把会员名录放到网上也是我们的目标之一）。

下面是HTML文档的基本框架：

<html>	← HTML文档的开始标记
<head>	← HTML文档头的开始标记
<title>My Page Title</title>	← HTML文档的标题
</head>	← HTML文档头的结束标记



```

<body bgcolor="white">          ← HTML文档内容的开始标记 (白色背景)
<h1>My Level 1 Heading</h1>    ← 一个一级标题

... content of document body ...

</body>                        ← HTML文档内容的结束标记
</html>                        ← HTML文档的结束标记

```

因为我们在前面的gen\_dir.pl脚本里已经搭建了一个可以灵活扩展的框架,所以用不着去编写一个全新的脚本来生成HTML格式的会员名录。只要我们能按HTML格式生成会员名录的有关代码插入到那个框架里,把会员名录放到网上的任务就算完成了。具体地说,我们需要在gen\_dir.pl脚本里增加以下几项内容:

- 编写HTML文档的初始化函数和扫尾函数。
- 编写一个用来对各有关数据项进行排版的函数。
- 在开关箱里增加一个用来标识HTML格式的元素,并把这个元素与用来生成HTML文档的那三个函数(初始化函数、排版函数、扫尾函数)关联起来。

上面给出的HTML文档基本框架可以很容易地划分为开头、中间、结束等三个部分。开头和结束部分对应着我们将要编写的初始化函数和扫尾函数,而中间部分则主要由排版函数来生成。HTML初始化函数将负责生成从HTML文档的开始标记直到一级标题之间的内容,而扫尾函数将负责生成</body>和</html>标记。如下所示:

```

sub html_init
{
    print "<html>\n";
    print "<head>\n";
    print "<title>U.S. Historical League Member Directory</title>\n";
    print "</head>\n";
    print "<body bgcolor=\"white\">\n";
    print "<h1>U.S. Historical League Member Directory</h1>\n";
}

sub html_cleanup
{
    print "</body>\n";
    print "</html>\n";
}

```

像往常一样,主要工作都将集中在排版函数里,但这部分代码也并不难编写:只要把gen\_dir.pl脚本里的format\_rtf\_entry()函数拷贝过来,再对会员记录项里的每一个特殊字符进行转义编码,最后再把RTF排版命令替换为HTML格式标记就行了。如下所示:

```

sub format_html_entry
{
    my $entry_ref = shift;

    # Convert <, >, ", and & to the corresponding HTML entities

```

```

# (&lt;;, &gt;;, &quot;, &amp;)
foreach my $key (keys (%{$entry_ref}))
{
    next unless defined ($entry_ref->{$key});
    $entry_ref->{$key} =~ s/&/&amp;/g;
    $entry_ref->{$key} =~ s/"/&quot;/g;
    $entry_ref->{$key} =~ s/>/&gt;/g;
    $entry_ref->{$key} =~ s/</&lt;/g;
}
printf "<strong>Name: %s</strong><br />\n", format_name ($entry_ref);
my $address = "";
$address .= $entry_ref->{street}
                if defined ($entry_ref->{street});
$address .= ", " . $entry_ref->{city}
                if defined ($entry_ref->{city});
$address .= ", " . $entry_ref->{state}
                if defined ($entry_ref->{state});
$address .= " " . $entry_ref->{zip}
                if defined ($entry_ref->{zip});
print "Address: $address<br />\n"
                if $address ne "";
print "Telephone: $entry_ref->{phone}<br />\n"
                if defined ($entry_ref->{phone});
print "Email: $entry_ref->{email}<br />\n"
                if defined ($entry_ref->{email});
print "Interests: $entry_ref->{interests}<br />\n"
                if defined ($entry_ref->{interests});
print "<br />\n";
}

```

下面是用这个排版函数生成的一个记录项:

```

<strong>Name: Mike Artel</strong><br />
Address: 4264 Lovering Rd., Miami, FL 12777<br />
Telephone: 075-961-0712<br />
Email: mike_artel@venus.org<br />
Interests: Civil Rights, Education, Revolutionary War<br />
<br />

```

遵照XHTML标准, 我们使用了<br />标记而非<br>标记。XHTML的语法比HTML更严格, 它们之间的主要区别将在第7.4.2节里介绍。

我们还需要对gen\_dir.pl脚本做最后一处修改: 在开关箱里增加一个与HTML格式相对应的新元素。下面是完成修改后的开关箱代码, 它的最后一个元素定义了一个名为html的排版格式, 这个元素将指向我们刚才编写的那几个用来生成HTML格式文档的函数:

```

# switchbox containing formatting functions for each output format
my %switchbox =
(

```

```

"banquet" =>                                # functions for banquet list
{
    "init"      => undef,                      # no initialization needed
    "entry"     => \&format_banquet_entry,
    "cleanup"   => undef                      # no cleanup needed
},
"rtf" =>                                       # functions for RTF format
{
    "init"      => \&rtf_init,
    "entry"     => \&format_rtf_entry,
    "cleanup"   => \&rtf_cleanup
},
"html" =>                                     # functions for HTML format
{
    "init"      => \&html_init,
    "entry"     => \&format_html_entry,
    "cleanup"   => \&html_cleanup
}
};

```

现在，只要执行下面这条命令就可以得到一份HTML格式的会员名录。此后，把directory.html文档添加到Web服务器的文档树里，就可以用Web浏览器来查看它了：

```
% ./gen_dir.pl html > directory.html
```

如果以后又对member数据表进行了修改，就需要再次运行这条命令以刷新在线版会员名录。如果不想以手动方式来执行这条命令，可以考虑把它设置为一项cron作业以定期执行在线版会员名录自动刷新。比如说，假设gen\_dir.pl脚本安装在/u/paul/bin目录里，“美国历史研究会”在Web服务器文档树里的目录是/usr/local/apache/htdocs/ushl，那么下面这条crontab设置项将在每天凌晨4点去自动刷新这份会员名录：

```
0 4 * * * /u/paul/bin/gen_dir.pl > /usr/local/apache/htdocs/ushl/directory.html
```

**注意** 如果想使用这个crontab设置项，就必须同时拥有/u/paul/bin/gen\_dir.pl脚本的执行权限和/usr/local/apache/htdocs/ushl/directory.html文档的写权限。

## 7.4 用DBI模块来开发Web应用

我们此前开发的DBI脚本都需要通过shell的命令行环境来使用，但DBI同样能用在其他上下文里，比如用在基于Web的应用程序开发中。当掌握了能够通过一个Web浏览器来调用的DBI脚本的编写技巧之后，你就能够以更加新颖和有趣的方式来使用数据库。比如说，可以把数据显示为表格的形式并把各数据列的名字设置为一个链接，只要点击一下这个链接，该数据列里的信息就将重新排序。还可以在Web网页里提供一个表单（form），让用户在这个表单里输入各种数据库检索条件，再把检索结果放到一个网页里返回给用户。这些做法虽然简单，却能大幅改善数据库访问操作的交互性，为数据库用户提供更多的方便。此外，因为Web浏览器的显示能

力通常要优于终端窗口，所以还能创造出更美观的输出效果来。

在这一节里，我们将编写以下几个基于Web的脚本：

- 为sampdb数据库编写一个通用的数据表浏览器：这个脚本与我们计划中的数据库应用任务没有任何关系。之所以要编写这个脚本，一是为了向大家演示Web程序设计工作中的几个概念，二是为了给大家提供一个方便的数据表内容查看工具。
- 为考试记分项目编写一个考试分数浏览器：这个脚本能让我们迅速查看到任何一次考试事件中的学生成绩。还可以利用它来确定评分曲线以评定学生们的学分。
- 为“美国历史研究会”编写一个用来查找兴趣相同的会员的脚本：这个脚本将根据会员们输入的搜索短语去搜索member数据表里的interests数据列，并把找到的记录项通过Web网页显示给会员。在前面的第7.3.4节里，我们曾经编写过一个能够完成这一任务的命令行脚本interests.pl，但那个命令行脚本必须在安装有它的机器上才能运行，分散在美国各地的会员很难享受到这一服务。而我们马上就要编写的基于Web的脚本却能实现这样一个目标：只要自己的机器里安装有一个Web浏览器，人们就能从“美国历史研究会”的在线会员名录里找到兴趣相同的朋友。此外，这两个脚本还可以互相验证，让人们在同一任务的不同完成办法有一个比较。（事实上，我们将开发两个基于Web的脚本实现：一个基于模式匹配（类似于interests.pl脚本的做法），另一个使用FULLTEXT索引进行搜索。）

编写这些脚本要用到Perl语言的CGI.pm模块，它提供了一套能够把DBI与Web联系起来的简单机制。（CGI.pm模块的获得和安装办法见附录A。）利用CGI.pm模块编写出来的DBI脚本能够使用对Web服务器与其他程序的编程接口做出了定义的CGI（Common Gateway Interface，通用网关接口）协议连接到Web服务器，这个模块的名字也正是由此而来。CGI.pm模块将负责完成各种辅助性的细节工作，比如接收Web服务器传递给脚本的参数值、生成正确的HTML排版标记等等。有了CGI.pm模块，就用不着再操心HTML排版标记方面的事情了，这大大减少了犯错误的机会——如果由你亲自去做这些事情的话，不仅要分散注意力，而且也容易出现写错排版标记或者排版标记不配对的错误。

这一章对CGI.pm模块的介绍已经足以让大家编写出自己的Web应用脚本，这里不可能把CGI.pm模块的方方面面都照顾到。如果想进一步了解这个模块的其他用法，可以去阅读由Lincoln Stein撰写的*Official Guide to Programming with CGI.pm*（CGI.pm模块编程指南，John Wiley公司1998年出版）一书或者查阅下面这个网址上的在线文档：

<http://stein.cshl.org/www/software/CGI/>

我的另外一本书*MySQL and Perl for the Web*（MySQL与Perl的网上应用，New Riders公司2000年出版），也是专门讨论MySQL与DBI程序设计问题的。

在本章后续内容里介绍的基于Web的脚本都可以在sampdb发行版本中的/perlapi/web目录里找到。

#### 7.4.1 配置Apache服务器来使用CGI脚本

要想开发基于Web的脚本，光有DBI和CGI.pm模块是不够的，还必须安装一个Web服务器

才行。这里的讨论重点将集中在Apache服务器与DBI脚本的配合使用方面,但只要你懂得变通并能灵活运用有关规则,也完全可以使用另外一种服务器。

在下面的讨论里,假设你们已经把Apache服务器软件安装在目录/usr/local/apache(如果使用的是UNIX系统)或者C:\Apache(如果使用的是Windows系统)里了。在Apache软件的顶级目录之下,有三个下级子目录与我们将要进行的开发工作关系最为密切,它们是htdocs(Apache服务器的HTML文档树)、cgi-bin(用来存放将由Apache服务器调用的可执行脚本和程序)和conf(用来存放Apache服务器的配置文件)。你的系统可能会把这些下级子目录安排在其他位置,如果真是这样的话,请对以下讨论内容做相应的调整。

cgi-bin目录不应该出现在Apache服务器的文档树里。这项安全措施有助于防止网络用户看到脚本的普通文本形式的源代码——你肯定不想让恶意用户接触到脚本的源代码并研究其中的安全漏洞。

如果你想让自己编写出来的脚本能够被Apache服务器调用,就需要把它拷贝到cgi-bin目录里去。在UNIX系统里,脚本的第一行必须以“#!”开头,脚本本身也必须被设置为可执行模式——就像一个命令行脚本一样。此外,最好把这个脚本的属主设置为将运行Apache服务器的用户并只允许这位用户进行访问。比如说,如果Apache服务器将被一个名为www的用户运行,下面两条命令将把脚本myscript.pl的属主设置为www并只允许这位用户执行和读取这个脚本:

```
# chown www myscript.pl
# chmod 500 myscript.pl
```

可能需要以root用户身份来执行这两条命令。如果你没有把脚本安装到cgi-bin目录里去的权限,请找系统管理员来帮忙做这件事。

如果使用的是Windows系统, chown和chmod命令就不需要了,但脚本的第一行仍需以“#!”开头。这一行用于列出Perl解释器程序的完整路径名。比如说,如果把Perl解释器安装为C:\Perl\bin\perl.exe,就应该把“#!”行写成下面这个样子(注意:Windows路径名中的反斜线字符“\”应该写成斜线字符“/”):

```
#! C:/Perl/bin/perl -w
```

在基于Windows NT的系统上,还有一个更简单的办法可供使用:把Perl解释器的路径名添加到环境变量PATH里。如果这样做,就可以把脚本的第一行写成下面这个样子:

```
#! perl -w
```

sampdb发行版本里的Perl语言脚本都在“#!”行里把Perl解释器的路径名写成“/usr/bin/perl”的样子。如果把Perl安装在其他地方,就需要对sampdb发行版本里的每一个Perl语言脚本的“#!”行做相应的修改。

把脚本安装到cgi-bin目录里之后,就可以通过从Web浏览器向Web服务器发出相应的URL地址的办法来请求它了。比如说,如果Web服务器运行在主机www.snake.net上,就可以使用下面这个URL地址来请求myscript.pl脚本:

```
http://www.snake.net/cgi-bin/myscript.pl
```

用Web浏览器来请求一个脚本将导致它被Web服务器调用执行,该脚本的执行结果将被Web



服务器送回到Web浏览器并显示为一个页面。

当从命令行执行DBI脚本的时候，警告和出错信息都将被送往终端。但因为Web环境根本不存在一个这样的终端，所以警告和错误信息（以及其他一些信息）都将被送往Apache服务器的错误日志。这些信息对脚本的调试工作有着重要的帮助意义，所以一定要把这个日志在系统里的存放地点找出来。在我的系统上，它是Apache安装目录/usr/local/apache下的logs目录里的error\_log文件。你系统上的情况可能与我的不同。这个日志的存放地点是由配置文件httpd.conf（可以在Apache服务器的conf目录里找到这个文件）里的ErrorLog选项设定的。

#### 7.4.2 CGI.pm模块简介

如果在编写Perl脚本的时候会用到CGI.pm模块，就必须在脚本的开头部分放上一条use CGI语句以导入这个模块里的函数名。下面这条语句将导入由CGI.pm模块中最常用的那些函数构成的标准集：

```
use CGI qw(:standard);
```

有了这条语句，就可以调用CGI.pm模块中的有关函数来生成各种HTML结构了。一般说来，那些函数的名字与相应的HTML排版标记是一致的。比如说，如果想生成一个一级标题和一个段落，就需要调用h1()和p()函数，如下所示：

```
print h1 ("This is a header");  
print p ("This is a paragraph");
```

CGI.pm模块还支持面向对象的编程风格，也就是说，可以在事先没有导入其函数名的情况下调用这个模块里的函数。这套方案的具体做法是：先像下面这样在脚本里增加一条use语句并创建一个CGI对象：

```
use CGI;  
my $cgi = new CGI;
```

然后就可以在脚本里通过这个CGI对象来调用CGI.pm模块里的函数了。此时，那些函数将被视为这个CGI对象的方法：

```
print $cgi->h1 ("This is a header");  
print $cgi->p ("This is a paragraph");
```

如果使用的是面向对象的接口方案，就必须总是写出“\$cgi->”这个前缀，为简洁起见，本书里将使用形式比较简单的函数调用接口方案。但使用函数调用接口方案有这样一个弊端：一旦CGI.pm模块里的函数与Perl语言中的内建函数发生同名现象，就不得不另想一种不会产生冲突的调用办法。比如说，CGI.pm模块里有一个名为tr()的函数，它将生成一对放在HTML表格各单元两端的<tr>和</tr>标记，这个函数与Perl语言中用来对字符串里的字符进行替换的内建函数tr出现了同名冲突。为了解决这一矛盾，就必须在使用CGI.pm模块的函数调用接口方案时把tr()写成Tr()或TR()。如果使用的是面向对象的接口方案，就不会出现这种问题：因为tr()是\$cgi对象的一个方法，所以在代码里得用\$cgi->tr()的写法来调用它，而这就能与Perl语言内建的函数名区分开了。

### 1. 检查Web输入参数

CGI.pm模块可以完成很多基础性的细节工作, 为脚本收集经Web服务器传递过来的输入信息就是其中之一。只要在脚本里调用param()函数就可以获得来自Web的输入, 根本用不着去考虑它们是怎样得来的。如果想知道Web服务器给脚本传递来哪些参数(即各参数的名字), 可以使用下面这条语句:

```
my @param = param ();
```

如果想检索特定参数的值, 把它的名字传递到param()函数里就行了。如果该参数有值, param()函数将返回它的值; 如果该参数没有值, 则返回undef。如下所示:

```
my $my_param = param ("my_param");
if (defined ($my_param))
{
    print "my_param value: $my_param\n";
}
else
{
    print "my_param is not set\n";
}
```

### 2. 编写Web输出

CGI.pm模块里的很多函数都是用于生成各种将被送往客户浏览器的输出信息。请看下面这段HTML文档:

```
<html>
<head>
<title>My Simple Page</title>
</head>
<body bgcolor="white">
<h1>Page Heading</h1>
<p>Paragraph 1.</p>
<p>Paragraph 2.</p>
</body>
</html>
```

下面这个脚本将使用CGI.pm输出函数生成一份与上面这段HTML文档相当的输出信息来:

```
#!/usr/bin/perl -w
# simple_doc.pl - produce simple HTML page

use strict;
use CGI qw(:standard);

print header ();
print start_html (-title => "My Simple Page", -bgcolor => "white");
print h1 ("Page Heading");
print p ("Paragraph 1.");
print p ("Paragraph 2.");
print end_html ();
```

header()函数负责生成一个“Content-Type:”标头,这个标头必须出现在网页的最开头。如果网页是通过一个脚本动态生成的,这个标头就必不可少,它的作用是让浏览器知道紧随其后的文档属于哪一种类型。(这与我们编写静态HTML网页时的做法稍有不同。静态HTML网页不需要有这个标头,因为Web服务器会自动发送一个这样的标头给浏览器。)在默认的情况下,header()函数将写出一个如下所示的标头来:

```
Content-Type: text/html
```

header()函数的后面就是那些用来生成网页内容的CGI.pm输出函数了。start\_html()函数负责生成从<html>开始直到<body>位置的各种HTML排版标记,h1()和p()函数负责写出一级标题和段落内容,end\_html()函数负责写出HTML文档最末尾的结束标记。

很多CGI.pm函数,比如刚才的start\_html()函数,都允许使用一些命名的参数,这些参数的使用格式都是“-name => value”。这种安排对带有很多可选参数的函数特别有利,因为不仅可以只列出需要用到的参数,还可以按任意顺序来列出它们。

虽然CGI.pm模块为我们提供了大量的输出生成函数,但仍可根据具体情况去自行生成一些HTML排版标记。可以把这两种做法结合起来使用,让脚本里既有对CGI.pm函数的调用语句,又有用来亲自生成某些HTML排版标记的Perl打印语句。不过,与通过Perl打印语句亲自写出HTML排版标记的做法相比,使用CGI.pm函数来生成输出信息的做法有助于把注意力集中到脚本代码的执行逻辑方面而非具体的排版标记上,同时,脚本生成的HTML也不太容易出现错误。(注意,这里说的是“不太容易出现错误”,因为CGI.pm模块本身并不能阻止你做出一些不合逻辑的事情,比如在标题里放上一个列表等等。)

CGI.pm模块还使脚本更具有可移植性,这是亲自写出HTML排版标记时很难做到的。比如说,从2.69版开始,CGI.pm将自动生成XHTML格式的输出。如果脚本是用老版本的CGI.pm模块编写出来的,那么,只需把CGI.pm模块升级到最新的版本,用那些脚本所生成的HTML文档也将自动从普通的HTML格式升级为XHTML格式。

XHTML与HTML很相似,但有一个更加明确定义的格式。HTML的要求不那么严格,所以很容易学习和使用,但同时也导致了很多问题。比如说,不同的浏览器对HTML往往有着不同的解释,对不规范的HTML文档也有着不同的容忍性——在这个浏览器里能正确显示的网页到了那个浏览器里往往就不能正确显示了。XHTML的要求就要严格得多了,这就使XHTML文档的格式更加规范。下面是HTML和XHTML的几个主要区别:

- 与HTML不同,XHTML要求文档中的每一个起始标记必须有一个配对的结束标记。比如说,段落本应该放在<p>和</p>标记之间,但HTML文档却往往允许省略</p>标记。在XHTML文档里,这个</p>标记必不可少。我们知道,有些HTML标记(比如<br>和<hr>)本身并不要求配对使用,但因为XHTML要求排版标记必须配对出现,所以往往会在XHTML文档里看到“<br></br>”和“<hr></hr>”等比较“丑陋”的写法。为解决这一问题,XHTML也允许使用单个的<br/>和<hr/>标记来同时充当起始标记和结束标记。不过,因为某些早期的浏览器不能正确地解释这类标记,所以最好把它们写成<br />和<hr />(即在斜线字符的前面增加一个空格)以减少这种错误的发生。

- HTML不区分排版标记和属性名中的字母大小写情况。比如说, HTML会把<tr>和<TR>看做是同样的标记。XHTML要求排版标记和属性名必须使用小写字母, 所以只能把这个标记写成<tr>的样子。
- HTML允许不把属性值放在引号中间, 甚至允许不写出属性值。比如说, 下面这样的写法在HTML里是合法的:

```
<td width=40 nowrap>Some text</td>
```

XHTML则要求必须有属性值, 而且属性值还必须放在引号中间。对于那些习惯于不带属性值使用的HTML属性, XHTML要求把它们的名字用做它们的值。在XHTML文档里, 必须把上面那个HTML标记写成下面的样子:

```
<td width="40" nowrap="nowrap">Some text</td>
```

本书里的Web脚本所生成的输出文档全都遵守XHTML规范。在这一章里, 我们将完全依靠CGI.pm模块去生成正确的XHTML标记。但第8章里的脚本就要靠它们自己去生成各种排版标记了, 这是因为PHP不像CGI.pm模块那样拥有提供XHTML排版标记的函数。

### 3. 对HTML和URL文本进行转义

如果准备写到Web页面里去的文本可能包含有特殊字符, 就应该使用escapeHTML()函数对这段文字进行处理以确保那些特殊字符都能得到正确的转义。如果构造出来的URL地址字符串可能包含有特殊字符, 那么也得对它进行必要的转义处理, 但此时应该使用escape()函数来做这件事。这两个函数的特殊字符是不同的, 它们用来对特殊字符进行编码的格式也不相同, 这就要求必须选用正确的编码函数来对有关字符进行转义处理。请看下面这个简短的Perl语言脚本escape\_demo.pl:

```
#!/usr/bin/perl -w
# escape_demo.pl - demonstrate CGI.pm output-encoding functions

use CGI qw(escapeHTML escape); # import escapeHTML() and escape()

$s = "x<=y, right?";
print escapeHTML ($s) . "\n"; # encode for use as HTML text
print escape ($s) . "\n";    # encode for use in a URL
```

这个脚本分别使用刚才介绍的那两个函数对字符串\$s进行了编码并把编码结果打印了出来。当运行这个脚本的时候, 它将产生如下所示的输出。我们可以清楚地看到, 字符串\$s作为HTML文本时的编码与它作为URL地址时的编码是不同的:

```
x&lt;=y, right?
x%3C%3Dy%2C%20right%3F
```

escape\_demo.pl脚本通过一条use CGI语句导入了编码函数的函数名。注意, CGI.pm模块的标准函数集并不包括这两个编码函数在内, 所以哪怕已经导入了CGI.pm模块的标准函数集, 也必须另外导入这两个函数。如下所示:

```
use CGI qw (:standard escapeHTML escape);
```



#### 4. 编写多用途网页

为什么要用基于Web的脚本来生成HTML页面而不去编写静态的HTML文档呢？一个重要的原因是脚本能够根据不同的调用情况生成不同的页面。我们后面将要编写的脚本都具有这一特性，它们都将按以下策略执行：

- 当从浏览器第一次请求某个脚本时，它会生成一个初始页面供你在该页面里选择想要查找的信息。
- 当做出选择后，浏览器将再向Web服务器发出一个请求，这个请求将导致刚才的那个脚本被再次调用执行。然后，这个脚本将把所请求的信息从数据库里检索出来并显示在第二个页面里。

要想实现这一构想，就必须解决这样一个问题：如何根据第一个页面里做出的选择来确定第二个页面里的内容。但Web页面通常是彼此无关的，要想让前、后两个页面发生关联，就必须做出特殊的安排。一种解决方案是让脚本在生成前一个页面的时候把某个参数设定为一个特定的值，用这个值来告诉脚本你想让它在下一次被调用的时候干些什么。当第一次调用这个脚本的时候，那个参数还没有值，于是脚本将生成一个初始页面，当在初始页面里做出选择之后，这个脚本将再次被调用。因为那个参数现在已经有了一个值，脚本就将根据选择生成另外一个页面。

可供Web页面用来向脚本传递参数值的办法有好几种。一种办法是在页面里提供一个表单来供用户填写。当提交这个表单时，它的内容就会提交到Web服务器。Web服务器再把有关信息传递给脚本，脚本通过调用param()函数就能知道提交给它的参数值是什么了。我们为“美国历史研究会”会员寻找兴趣相同的朋友而进行的搜索工作就将用这个办法来实现：在搜索页面里安排一个表单，由会员在表单里填写他们想进行搜索的关键字。

向脚本传递参数值的另一种办法是把参数值追加在URL地址字符串的尾部，当你向Web服务器请求该URL地址处的脚本时，那些参数值也将被传递给Web服务器。我们用来实现sampdb数据表浏览器和考试分数浏览器的脚本将用这个办法来实现。这个办法的工作过程是这样的：在某脚本生成的页面里包含着一些超链接（hyperlink，也叫做超文本链接），这些超链接分别带有一个用来告诉脚本你想让它做什么事情的参数值，当点击某个超链接的时候，这个脚本就会被再次调用执行（实际上，这等于是让脚本自己去调用自己），并根据追加在URL地址字符串末尾的参数值（也就是具体点击的是哪一个超链接）去决定将要生成什么样的页面。

如果让某个脚本去调用自己，就需要在该脚本发送给浏览器的页面里加上一个指向它自身的超链接，即一个指向它自己的URL地址的链接。比如说，假设脚本myscript.pl已经被安装到了Web服务器的cgi-bin目录，并且这个脚本所生成的页面里有一个如下所示的链接：

```
<a href="/cgi-bin/myscript.pl">Click Me!</a>
```

那么，当在浏览器里点击该页面中的文字“Click Me！”时，浏览器就会向Web服务器发回一条执行myscript.pl脚本的请求。当然，因为上例中的URL地址没有附带任何其他的信息，所以它只能做到让脚本再次送回同样的页面。可是，如果还给它加上了一个参数，那个参数也将在点击这个链接的时候送回给Web服务器。接下来，当Web服务器调用这个脚本的时候，这个脚本就可以通过param()函数检测到该参数的取值情况并据此采取相应的行动。



把一个参数值追加到URL地址字符串末尾的办法是：先写出一个问号字符“?”，再以“*name = value*”的形式给出这个参数的名字和值。比如说，如果想把一个名字是“size”、值是“large”的参数追加到上例中的URL地址字符串的末尾，可把整个URL地址字符串写成如下所示的样子：

```
/cgi-bin/myscript.pl?size=large
```

如果需要附加多个参数，就需要用分号字符“;”<sup>①</sup>作为它们之间的分隔符。如下所示：

```
/cgi-bin/myscript.pl?size=large;color=blue
```

如果想在脚本里构造出一个带有参数的自引用URL地址，就需要在脚本里先使用url()函数得到它自身的URL地址，再把有关参数追加到这个地址的末尾，如下所示：

```
$url = url ();          # get URL for script
$url .= "?size=large";  # add first parameter
$url .= ";color=blue";  # add second parameter
```

之所以要调用url()函数来获得脚本的路径，是为了避免把脚本的路径硬编码在代码里。

接着，把这个URL地址送入CGI.pm模块中的a()函数以生成一个指向该脚本自身的超链接：

```
print a ({-href => $url}, "Click Me!");
```

这条print语句将生成一个如下所示的超链接：

```
<a href="/cgi-bin/url.pl?size=large;color=blue">Click Me!</a>
```

上面这个例子在构造\$url字符串的时候并没有考虑到参数值或链接标签（即链接本身的文字内容）里可能会有特殊字符的情况。为保险起见，还是用CGI.pm模块所提供的编码函数对那些参数值和链接标签进行一下处理为好——除非能百分之百地肯定不需要对它们进行任何编码。将出现在URL地址字符串里的值应该用escape()函数来编码，将出现在HTML文本里的值应该用escapeHTML()函数来编码。就拿上面那个例子来说，如果把超链接标签的值保存在\$label变量里，把参数size和color的值保存在\$size和\$color变量里，就应该像下面这样对它们进行一下编码处理：

```
$url = sprintf ("%s?size=%s;color=%s",
                url (), escape ($size), escape ($color));
print a ({-href => $url}, escapeHTML ($label));
```

下面这个简短的CGI脚本flip\_flop.pl，演示了自引用URL地址的使用方法。在这个脚本第一次被调用时生成的页面（我们不妨称之为“页面A”）里有一个指向该脚本自身的超链接，这个超链接又带有一个名为pageb的参数。点击页面A里的超链接将导致Web服务器再次调用flip\_flop.pl脚本并生成另一个页面——我们称之为“页面B”。页面B里也有一个指向该脚本自身的超链接，但这个超链接不带pageb参数。（在这个例子里，参数pageb的取值是什么并不重要，

① CGI.pm模块还能把URL地址字符串中的“&”字符识别为参数分隔符。不同程序设计语言的Web编程API有着不同的记号体系，所以大家在构造URL地址字符串的时候一定要把它们能够识别的分隔符是“;”还是“&”搞清楚。

重要的是有没有这个参数。)这就意味着如果点击了页面B里的超链接,就会重新看到页面A。换句话说,连续调用flip\_flop.pl脚本将使它交替生成页面A和页面B:

```
#! /usr/bin/perl -w
# flip_flop.pl - simple multiple-output-page CGI.pm script

use CGI qw(:standard);

my $url;

# determine which page to display based on absence or presence
# of the pageb parameter

if (!defined (param ("pageb"))) # display page A w/link to page B
{
    $this_page = "A";
    $next_page = "B";
    $url = url () : "?pageb=1";
}
else # display page B w/link to page A
{
    $this_page = "B";
    $next_page = "A";
    $url = url ();
}

print header ();
print start_html (-title => "Flip-Flop: Page $this_page",
                  -bgcolor => "white");
print p ("This is Page $this_page. To select Page $next_page, "
        . a ({-href => $url}, "click here"));
print end_html ();
```

把这个脚本安装到cgi-bin目录,再从浏览器里用下面这个URL地址请求它:

[http://www.snake.net/cgi-bin/flip\\_flop.pl](http://www.snake.net/cgi-bin/flip_flop.pl)

在交替出现的页面A和页面B里的超链接上多点击几次,看看flip\_flop.pl脚本到底是如何来交替生成这两个页面的。

现在,如果有另外一个客户也开始请求flip\_flop.pl脚本,会发生什么呢?两个客户会彼此干扰吗?答案是不会。因为两个客户在首次请求这个脚本的时候都没有给出pageb参数,所以两个客户首次看到的页面将都是这个脚本所生成的页面A。而两个客户此后发出的请求将分别根据各自看到的当前页面而带有或者不带有pageb参数,所以flip\_flop.pl脚本也将为两个客户分别生成相应的页面而根本不会干扰到另外一个客户。

### 7.4.3 从Web脚本连接MySQL服务器

在第7.3节里开发了一些命令行脚本,那些脚本在与MySQL服务器建立连接的时候使用的是

同一段共享代码。下面将要编写的CGI脚本也将使用一段共享代码来建立与MySQL服务器的连接,但这两段共享代码还是有一些差异的:

```
#! /usr/bin/perl -w
use strict;
use DBI;
use CGI qw(:standard);

use Cwd;
# option file that should contain connection parameters for UNIX
my $option_file = "/usr/local/apache/conf/sampdb.cnf";
my $option_drive_root;
# override file values for Windows
if ($^O =~ /^MSWin/i || $^O =~ /^dos/)
{
    $option_drive_root = "C:/";
    $option_file = "/Apache/conf/sampdb.cnf";
}
# construct data source and connect to server (under Windows, save
# current working directory first, change location to option file
# drive, connect, then restore current directory)
my $orig_dir;
if (defined ($option_drive_root))
{
    $orig_dir = cwd ();
    chdir ($option_drive_root)
        or die "Cannot chdir to $option_drive_root: $!\n";
}
my $dsn = "DBI:mysql:sampdb:mysql_read_default_file=$option_file";
my $dbh = DBI->connect ($dsn, undef, undef,
                        { RaiseError => 1, PrintError => 0 });
if (defined ($option_drive_root))
{
    chdir ($orig_dir)
        or die "Cannot chdir to $orig_dir: $!\n";
}
```

上面这段代码与我们在前面的命令行脚本里使用的同功能代码有以下几点差异:

- 上面这段代码的开头部分包含有use CGI和use Cwd两条语句。第一条语句用来导入CGI.pm模块。第二条语句所导入的模块将用来切换当前工作目录的路径名,它将使我们的脚本在Windows环境下也能正确执行。
- 上面这段代码不再对来自命令行的连接参数进行处理——我们将把这些参数列在一个选项文件里。
- 上面这段代码没有使用mysql\_read\_default\_group来读取标准选项文件。我们将使用mysql\_read\_default\_file来只读取一个专为sampdb数据库而准备的选项文件,换句话说,我们将要编写的Web脚本都将只使用这个选项文件里的连接参数来访问sampdb数据库。正

如大家看到的那样，上面这段代码将从/usr/local/apache/conf/sampdb.cnf文件（如果使用的是UNIX系统）或C:\Apache\conf\sampdb.cnf文件（如果使用的是Windows系统）读取连接参数。在Windows系统上，上面这段代码会在读取选项文件之前先把当前路径切换到选项文件所在的硬盘根目录，等读取完选项文件之后再把当前路径切换回原来的目录。至于为什么要做这种路径切换，已经在第7.2.9节里解释过了。

在sampdb发行版本里有一份现成的sampdb.cnf文件，可以把它直接安装到系统里供那些基于DBI的Web脚本使用。它的内容如下所示：

```
[client]
host=cobra.snake.net
user=sampadm
password=secret
```

如果你想在自己的系统上试用这一章里开发的基于Web的脚本，请把这个选项文件的名字（即sampdb.cnf）修改为你打算使用的文件名。你可能还需要在选项文件里把有关参数修改为你将要使用的MySQL服务器主机名以及你的MySQL账户名和口令。

如果你使用的是UNIX系统，那还应该把这个选项文件的属主设置为将用来运行Apache服务器的那个账户并把这个文件的访问模式设置为400或600以阻止其他用户读取这个选项文件。如果不这样做，在你的Web服务器主机上有登录账户的其他用户就能直接读取这个选项文件的内容，这绝对是一个系统安全漏洞。

然而，即使这样做了，那些有权在你的Web服务器上安装可执行脚本的其他用户也仍能读出这个选项文件的内容。我们知道，被Web服务器调用执行的脚本拥有着用来运行Web服务器的那个账户的所有权限。也就是说，有权安装Web脚本的其他用户完全能够编写并执行一个这样的脚本：它将在运行时打开你的选项文件并把里面的内容显示在一个Web页面里。因为那个脚本拥有着用来运行Web服务器的那个账户的所有权限，所以它完全有权去读取你的选项文件——你用来连接MySQL服务器和访问sampdb数据库的连接参数将暴露无遗。如果你信不过系统上的其他用户，就应该考虑采取这样一种对策：先创建一个在sampdb数据库上只有只读（SELECT）权限的MySQL账户，再把这个新建账户的用户名和口令（注意：不是你本人的MySQL用户名和口令）列在sampdb.cnf选项文件里。这样，即使别人看到了这个选项文件的内容，你的损失也只是被别人看到sampdb数据库里的东西而已——因为别人不能通过一个有权修改数据表的MySQL账户（即你的账户）连接到你的数据库去。创建一个权限受到限制的MySQL用户账户的办法见第11章。这一策略的缺陷是：因为你使用的是一个只拥有只读权限的MySQL账户，所以即使是你本人编写出来的脚本也只能进行数据检索，而不能进行数据录入。

另一种办法是设法在Apache的suEXEC机制下执行你的脚本。其具体做法是：先选定你的脚本将以哪一个可信任用户的身份来运行，再把选项文件的属主设置为那个可信任用户并只允许该用户来读取，最后再编写你的脚本必须从这个选项文件里读取连接参数。还有一种只适用于Apache 2.x版本的办法：先把你的脚本放到某个特定的目录里——Apache 2.x服务器能够根据脚本所在的目录以特定用户（比如你本人）的权限来运行它们，再把sampdb.cnf文件和各有关脚本的属主及访问模式设置为只允许你一个人去访问，这样，其他用户就无法染指它们了。

再有一种办法是让你的脚本要求由使用者提供一个用户名和口令，再用这个用户名和口令去连接MySQL服务器。这个办法比较适用于那些使用频率不高的系统管理类脚本，用在日常使用的脚本里往往会让人觉得比较麻烦。而且，万一有人在Web服务器与浏览器之间的网络上偷偷地放上了一个嗅探器的话，这种要求由使用者提供一个用户名和口令的做法反而容易发生泄密，所以你很可能不得不专门为此去建立一个安全连接——这超出了本书的讨论范围。

从上面这些讨论我们不难得出这样一个结论：Web脚本的安全性是一个非常复杂的问题。这个问题的涉及面是如此之广，复杂性是如此之高，所以除了希望大家能够自己去加强这方面的学习外，很难再给出更好的建议了。在前面提到的*MySQL and Perl for the Web*（MySQL与Perl的网上应用，New Riders公司2000年出版）里有一章是专门讨论网络安全问题的，大家可以在那里查到如何使用SSL来建立安全连接。Apache服务器的使用手册里也有不少关于网络安全方面的讨论，下面这个网址上的网络安全FAQ（常见问题解答）也非常值得一读：

<http://www.w3.org/Security/Faq/>

#### 7.4.4 基于Web的数据库浏览器

我们的第一个基于Web的应用程序是一个简单的脚本db\_browse.pl，它不仅能查出sampdb数据库里现有哪些数据表，而且能让你从Web浏览器上交互地查看这些数据表中的内容。这个脚本的工作情况是这样的：

- 当从浏览器第一次请求db\_browse.pl脚本的时候，它将连接到MySQL服务器，检索出sampdb数据库里的现有数据表并显示在一个页面里，这个页面里列出的每个数据表都是一个超链接。当在页面里点击某个数据表名链接时，浏览器将向Web服务器发出一个要求db\_browse.pl脚本显示该数据表内容的请求。
- 如果db\_browse.pl脚本在它被调用的时候发现你选中了某个数据表的名字，它就会把这个数据表的内容检索出来并把这些信息显示在你的Web浏览器里。数据表各数据列的名字将被显示为超链接，如果点击了某个数据列的名字，浏览器就会向Web服务器发出一个重新显示该数据表内容的请求，但这一次有关信息将按所选定的那个数据列进行排序。

在继续学习之前，先提醒大家一句：虽然db\_browse.pl脚本演示了很多Web程序设计中的重要概念，但它本身却可能会成为系统上的一个安全漏洞，给站点上的不友好访客以可乘之机。恶意访客能够利用这个脚本轻易地查看到你在sampdb.cnf文件里给出的那个MySQL用户所能访问的任何一个数据表的内容，其具体做法稍后介绍。就眼下来说，建议你把这个脚本安装在一个只能访问非敏感数据的MySQL账户下，并在你试用过它并了解了它的工作原理后立刻把它移出cgi-bin目录。（另一种办法是把这个脚本安装在一个非信任用户无法访问的私用服务器上。）为什么说db\_browse.pl脚本是一个安全漏洞呢？请看下面这个例子。我们将在第8章编写一个能够让“美国历史研究会”会员用来通过Web编辑其会员记录的脚本，这个脚本对会员记录项的访问是由存放在member\_pass数据表里的口令来控制的。如果那时还把db\_browse.pl脚本留在cgi-bin目录里，任何人都能利用db\_browse.pl脚本查看到这个口令数据表里的内容——那些修改member数据表里的记录项所必需的口令信息将暴露无遗。



好了，如果你还没有被上面这段文字吓倒的话，下面我们一起去看看db\_browse.pl脚本的代码。这个脚本先生成Web页面的开头部分，然后检查tbl\_name参数看是否需要显示某个特定的数据表：

```
#!/usr/bin/perl -w
# db_browse.pl - Allow sampdb database browsing over the Web

use strict;
use DBI;
use CGI qw (:standard escapeHTML escape);

# ... set up connection to database (not shown) ...

my $db_name = "sampdb";

# put out initial part of page
my $title = "$db_name Database Browser";
print header ();
print start_html (-title => $title, -bgcolor => "white");
print h1 ($title);

# parameters to look for in URL
my $tbl_name = param ("tbl_name");
my $sort_col = param ("sort_col");

# If $tbl_name has no value, display a clickable list of tables.
# Otherwise, display contents of the given table. $sort_col, if
# set, indicates which column to sort by.

if (!defined ($tbl_name))
{
    display_table_names ($dbh, $db_name)
}
else
{
    display_table_contents ($dbh, $tbl_name, $sort_col);
}

print end_html ();
```

有了CGI.pm模块，Web服务器传递给脚本的参数值就很容易处理了，只需把参数名送入param()函数，就能获得它的值。db\_browse.pl脚本首先判断tbl\_name参数是否有值：如果没有值，就说明这是脚本的第一次调用，脚本就会生成一份数据表清单；如果有值，脚本就会把tbl\_name参数指定的数据表的内容按sort\_col参数指定的数据列排序后显示出来。

display\_table\_names()函数负责生成初始页面。display\_table\_names()函数检索数据表清单并且生成的HTML输出是一个子弹式清单，清单里的每一项是sampdb数据库里的某个数据表的名字：

```

sub display_table_names
{
    my ($dbh, $db_name) = @_;

    print p ("Select a table by clicking on its name:");

    # retrieve reference to single-column array of table names
    my $ary_ref = $dbh->selectcol_arrayref (qq{ SHOW TABLES FROM $db_name });

    # Construct a bullet list using the ul() (unordered list) and
    # li() (list item) functions. Each item is a hyperlink that
    # re-invokes the script to display a particular table.
    my @item;
    foreach my $tbl_name (@{$ary_ref})
    {
        my $url = sprintf ("%s?tbl_name=%s", url (), escape ($tbl_name));
        my $link = a ({-href => $url}, escapeHTML ($tbl_name));
        push (@item, li ($link));
    }
    print ul (@item);
}

```

li()函数负责生成每个清单项两端的<li>和</li>标记, ul()函数负责生成一组清单项两端的<ul>和</ul>标记。清单里列出的每一个数据表名都被设置为一个超链接, 点击某个超链接将导致Web服务器再次调用db\_browse.pl脚本去显示有关数据表的内容。display\_table\_names()函数生成的HTML输出如下所示:

```

<ul>
<li><a href="/cgi-bin/localhost/db_browse.pl?tbl_name=absence">absence</a></li>
<li><a href="/cgi-bin/localhost/db_browse.pl?tbl_name=event">event</a></li>
<li><a href="/cgi-bin/localhost/db_browse.pl?tbl_name=member">member</a></li>
...
</ul>

```

如果tbl\_name参数在db\_browse.pl脚本被调用的时候有值, 脚本就会把这个值连同参数sort\_col的值(脚本将按sort\_col参数指定的数据列对数据库查询结果进行排序)传递到display\_table\_contents()函数里去:

```

sub display_table_contents
{
    my ($dbh, $tbl_name, $sort_col) = @_;
    my @rows;
    my @cells;

    # if sort column not specified, use first column
    $sort_col = "1" if !defined ($sort_col);

    # present a link that returns user to table list page

```

```

print p (a ({-href => url ()}, "Show Table List"));

print p (strong ("Contents of $tbl_name table:"));

my $sth = $dbh->prepare (qq{
    SELECT * FROM $tbl_name ORDER BY $sort_col
    LIMIT 200
});
$sth->execute ();

# Use the names of the columns in the database table as the
# headings in an HTML table. Make each name a hyperlink that
# causes the script to be reinvoked to redisplay the table,
# sorted by the named column.

foreach my $col_name (@{$sth->{NAME}})
{
    my $url = sprintf ("%s?tbl_name=%s;sort_col=%s",
                        url (),
                        escape ($tbl_name),
                        escape ($col_name));
    my $link = a ({-href => $url}, escapeHTML ($col_name));
    push (@cells, th ($link));
}
push (@rows, Tr (@cells));

# display table rows
while (my @ary = $sth->fetchrow_array ())
{
    @cells = ();
    foreach my $val (@ary)
    {
        # display value if non-empty, else display non-breaking space
        if (defined ($val) && $val ne "")
        {
            $val = escapeHTML ($val);
        }
        else
        {
            $val = "&nbsp;";
        }
        push (@cells, td ($val));
    }
    push (@rows, Tr (@cells));
}

# display table with a border
print table ({-border => "1"}, @rows);
}

```

如果sort\_col参数没有值(即没有指定按哪个数据列进行排序), display\_table\_contents()函数将在数据库查询语句的末尾加上一条“ORDER BY 1”子句,也就是按数据表里的第一个数据列进行排序。我们在数据库查询命令里还使用了一条“LIMIT 200”子句,即把数据库查询命令返回的数据行个数限制为200条,这是为避免出现脚本发送大量数据到Web浏览器的情况而采取的预防措施。(sapmdb数据库里的数据表都没有200条以上的记录项。但如果用这个脚本去查看其他数据库里的数据表内容,这个预防措施就能发挥作用了。) display\_table\_contents()函数将把数据表里的数据行显示在一个HTML表格里:它用th()和td()函数来生成HTML表格的表头和表格单元,用Tr()函数把一组表格单元生成为一个HTML表格行,再用table()函数来生成HTML表格两端的<table>和</table>标记。

在脚本生成出来的HTML表格里,各列的标题其实都是一些超链接,当点击这些超链接的时候,脚本被再次调用并重新显示那个数据库表。这些超链接都带有一个sort\_col参数,这个参数将使脚本按明确指定的数据列对数据表内容进行排序。比如说,在用来显示event数据表内容的页面里,HTML表格列标题所对应的超链接是以下几个:

```
<a href="/cgi-bin/db_browse.pl?tbl_name=event&sort_col=date">date</a>
<a href="/cgi-bin/db_browse.pl?tbl_name=event&sort_col=type">type</a>
<a href="/cgi-bin/db_browse.pl?tbl_name=event&sort_col=event_id">event_id</a>
```

我们在display\_table\_contents()函数里还使用了这样一个技巧:把空值转换为一个不间断空格(&nbsp;)。这是因为这个脚本将要生成的是一个带有边框的HTML表格,如果HTML表格里有空值,有些浏览器就不能显示正确的边框,把空表格值转换为一个不间断空格就能解决这个问题。

如果想编写一个更通用的脚本,可以修改db\_browse.pl脚本以使它能够浏览多个数据库。比如说,可以在脚本里增加一些代码,让它在初始页面里列出一份服务器上的数据库清单而不是列出特定数据库(即sampdb数据库)里的数据表清单。等选择了某个数据库后,它再把该数据库里的数据表列在一个清单里。

db\_browse.pl脚本将通过sampdb账户去连接MySQL服务器。在本小节的开头部分,曾提到这个脚本很容易被别人利用来查看sampdb账户有权访问的任何一个数据表。为说明这一问题,我们不妨假设sampdb账户不仅能用来访问sampdb数据库,还能用来访问存放着公司人力资源信息(比如员工们的个人资料)的hr数据库——这就是一个安全漏洞。在构造URL地址字符串的时候,db\_browse.pl脚本使用的是sampdb数据库里数据表的名字,但这并不能阻止别人直接构造并发送出一个下面这样的URL地址去请求hr数据库里的某个数据表:

```
http://www.snake.net/cgi-bin/db_browse.pl?tbl_name=hr.employee
```

如果发生这种情况,虽然db\_browse.pl脚本仍将以sampdb账户连接到MySQL服务器并把sampdb数据库设置为默认数据库,但它构造出来的SELECT语句却会去访问hr数据库里的employee数据表,如下所示:

```
SELECT * FROM hr.employee ORDER BY 1
```

也就是说,只要能访问到你的Web服务器,别人就能利用db\_browse.pl脚本中的安全漏洞看到一个用来保存敏感信息的数据表(比如这里的employee数据表)里的内容。

### 7.4.5 考试记分项目：考试分数浏览器

我们的下一个脚本score\_browse.pl, 是设计用来显示考试记分项目里的学生们的考试分数的。严格地讲, 我们得先把考试分数录入到各有关数据表里才能对它们进行检索。不过, 因为用来录入考试分数的脚本将在下一章才会讲到, 所以现在只好利用前面各章零星录入的考试分数来将就一下了。也就是说, 虽然我们还没有一个方便的分数录入手段, 但将要编写的脚本还是有东西可供检索的。这个脚本将把考试分数显示为一个排好了序的清单, 这为确定评分曲线并给学生评定学分的工作提供了方便。

score\_browse.pl脚本与同为信息浏览器的db\_browse.pl脚本有很多相似之处, 但它的用途要更具体——查看学生们的某次考试或测验分数。它生成的初始页面是一个考试事件清单, 当你选中某次考试事件时, 就可以查看到学生们在那次考试或测验中的分数了。学生们的考试分数将按分数由高到低的顺序进行排序, 可以利用这个结果来确定评分曲线。

score\_browse.pl脚本只需检查event\_id这一个参数就能确定将要显示哪次考试事件的成绩。如果这个参数没有值, score\_browse.pl脚本将把event数据表里的数据行显示出来供用户从中选择一个; 如果这个参数有值, 它就会把与选中事件相关联的考试分数排序后显示出来:

```
# ... set up connection to database (not shown) ...

# put out initial part of page
my $title = "Grade-Keeping Project -- Score Browser";
print header ();
print start_html (-title => $title, -bgcolor => "white");
print h1 ($title);

# parameter that tells us which event to display scores for
my $event_id = param ("event_id");

# if $event_id has no value, display the event list.
# otherwise display the scores for the given event.
if (!defined ($event_id))
{
    display_events ($dbh)
}
else
{
    display_scores ($dbh, $event_id);
}

print end_html ();
```

display\_events()函数负责提取event数据表里的信息并把它们显示在一个HTML表格里, HTML表格列的标题就是数据库查询语句中给出的数据列的名字。在HTML表格的每一行上, 代表各次考试事件的event\_id值将被设置为一个超链接, 点击这个超链接, 就能看到学生们在这次考试中的分数。与各次考试事件相对应的URL地址由score\_browse.pl脚本的路径再加上一个用



来给出考试事件编号的参数构成, 如下所示:

```
/cgi-bin/score_browse.pl?event_id=n
```

下面是display\_events()函数的代码:

```
sub display_events
{
    my $dbh = shift;
    my @rows;
    my @cells;

    print p ("Select an event by clicking on its number:");

    # get list of events
    my $sth = $dbh->prepare (qq{
        SELECT event_id, date, type
        FROM event
        ORDER BY event_id
    });
    $sth->execute ();

    # use column names for table column headings
    for (my $i = 0; $i < $sth->{NUM_OF_FIELDS}; $i++)
    {
        push (@cells, th (escapeHTML ($sth->{NAME}->{$i})));
    }
    push (@rows, Tr (@cells));

    # display information for each event as a row in a table
    while (my ($event_id, $date, $type) = $sth->fetchrow_array ())
    {
        @cells = ();
        # display event ID as a hyperlink that reinvokes the script
        # to show the event's scores
        my $url = sprintf ("%s?event_id=%s", url (), escape ($event_id));
        my $link = a ({-href => $url}, escapeHTML ($event_id));
        push (@cells, td ($link));
        # display event date and type
        push (@cells, td (escapeHTML ($date)));
        push (@cells, td (escapeHTML ($type)));
        push (@rows, Tr (@cells));
    }

    # display table with a border
    print table ({-border => "1"}, @rows);
}
```

当选中某次考试事件时, 浏览器将送出一条调用score\_browse.pl脚本的请求, 这个请求的末

尾带有该次考试事件的编号。score\_browse.pl脚本再通过display\_scores()函数把学生们在event\_id参数所指定的那次考试中的分数显示出来。这个函数还会显示一个文本为“Show Events List”(显示考试事件清单)的超链接,点击这个超链接将返回初始页面,这样,就能迅速返回到考试事件清单页面并做出下一次选择了。下面是display\_scores()函数的代码:

```
sub display_scores
{
my ($dbh, $event_id) = @_;
my @rows;
my @cells;

# Generate a link to the script that does not include any event_id
# parameter. If the user selects this link, the script will display
# the event list.
print p (a ({-href => url ()}, "Show Event List"));

# select scores for the given event
my $sth = $dbh->prepare (qq{
    SELECT
        student.name, event.date, score.score, event.type
    FROM
        student, score, event
    WHERE
        student.student_id = score.student_id
        AND score.event_id = event.event_id
        AND event.event_id = ?
    ORDER BY
        event.date ASC, event.type ASC, score.score DESC
});
$sth->execute ($event_id); # bind event ID to placeholder in query

print p {strong ("Scores for event $event_id")};

# use column names for table column headings
for (my $i = 0; $i < $sth->{NUM_OF_FIELDS}; $i++)
{
    push (@cells, th (escapeHTML ($sth->{NAME}->[$i])));
}
push (@rows, Tr (@cells));

while (my @ary = $sth->fetchrow_array ())
{
    @cells = ();
    foreach my $val (@ary)
    {
        # display value if non-empty, else display non-breaking space
        if (defined ($val) && $val ne "")
```

```

        {
            $val = escapeHTML ($val);
        }
        else
        {
            $val = "&nbsp;";
        }
        push (@cells, td ($val));
    }
    push (@rows, Tr (@cells));
}

# display table with a border
print table ({-border => "1"}, @rows);
}

```

display\_scores()函数执行的数据库查询命令与第1.4.8节的“从多个数据表检索信息”小节里介绍JOIN命令的使用方法时给出的一个查询命令很相似。在第1章里，我们是根据一个给定的日期值来检索考试分数的，因为日期值要比抽象的event\_id编号值更容易理解。可在这个score\_browse.pl脚本里，我们却是根据一个event\_id编号值来检索考试分数的。之所以这样做，并不是因为我们现在已经知道event\_id编号是干什么用的了，而是因为脚本已经在初始页面里把这个编号（以及与之相关的考试日期和考试类型）显示出来并等待我们去选择了。说得更明白一点，这种操作接口使我们不必知道具体的细节就能完成想做的事情。拿眼前这个例子来说，我们用不着知道考试事件ID到底有什么含义——只要score\_browse.pl脚本知道这个编号的用途就足够了；我们只负责点击某个超链接，而把考试事件与日期相关联、把事件编号传递回Web服务器等具体的细节则全部由这个脚本代劳了。

#### 7.4.6 美国历史研究会：查找兴趣相同的会员

db\_browse.pl和score\_browse.pl脚本采用的是这样一种信息传递机制：通过一系列超链接向脚本的使用者提供一组选择项，当使用者做出选择时，被选中的超链接就将以预定的参数值再次调用这个脚本。Web脚本用来在自身与使用者之间传递信息的另一种办法是在Web页面里提供一个供使用者填写的HTML表单，当选择范围比较大或者无法预先确定各有关参数的值时，这种办法无疑更为适用。我们的下一个脚本就将使用这种机制来接受来自其使用者的输入。

在第7.3节里，我们曾编写过一个名为interests.pl的脚本来为“美国历史研究会”的会员查找有共同兴趣的其他会员。不过，那个脚本很难做到让每个会员人手一份，所以就必须由研究会的秘书（也就是你）从命令行去执行这个脚本，然后再把查询结果邮寄给提出这种要求的会员。要是能把这种查找功能的可用范围扩大到每一位会员，让他们都能自己去寻找与自己有着共同兴趣的其他会员，那可就好了。这个目标并不难实现，而基于Web的脚本恰好就是它的解决方案之一。本小节将向大家介绍两种数据表搜索技术：第一种以模式匹配操作为基础；第二种则使用了MySQL数据库提供的FULLTEXT搜索功能。

## 1. 利用模式匹配操作来进行搜索

我们将要编写的第一个搜索脚本叫做ushl\_browse.pl，它将在自己生成的HTML页面里提供一个表单供用户输入一个关键字。当用户提交这个表单的时候，Web服务器将再次调用这个脚本到member数据表里去搜索符合条件的会员并把搜索结果送回Web浏览器。这个脚本的数据表搜索机制是这样的：先在输入的关键字的两端分别加上一个“%”通配符，然后进行LIKE模式匹配，把member数据表里那些在interests数据列里包含有这个关键字的记录项找出来。

这个脚本的主要工作有两项：一是显示关键字表单；二是检查是否提交了关键字，如果是，则按上面介绍的流程进行一次搜索：

```
my $title = "U.S. Historical League Interest Search";
print header ();
print start_html (-title => $title, -bgcolor => "white");
print h1 ($title);

# parameter to look for
my $keyword = param ("keyword");

# Display a keyword entry form. In addition, if $keyword is defined,
# search for and display a list of members who have that interest.

print start_form (-method => "POST");
print p ("Enter a keyword to search for:");
print textfield (-name => "keyword", -value => "", -size => 40);
print submit (-name => "button", -value => "Search");
print end_form ();

# connect to server and run a search if a keyword was specified
if (defined ($keyword) && $keyword != /\s*$/)
{
    # ... set up connection to database (not shown) ...
    search_members ($dbh, $keyword);
    # ... disconnect (not shown) ...
}
```

这个脚本用来向它自己传递信息的办法与db\_browse.pl或score\_browse.pl脚本里的做法不同。在这个脚本里，根本没有把keyword参数追加到一个URL地址末尾的代码。相反，表单里的信息将由Web浏览器负责进行编码，然后再由浏览器把编码后的信息夹杂在一个POST请求里送回Web服务器。不过，不管信息是以何种方式送回Web服务器的，CGI.pm模块中的param()函数都能一视同仁地把想要的参数值提取出来——就好像这些方式根本没有区别似的，这是CGI.pm模块在简化Web程序设计工作方面的又一大贡献。

关键字的搜索工作由search\_members()函数负责完成。它以一个数据库句柄和你在Web页面里输入的关键字为输入参数，然后运行数据库检索命令并把找到的会员记录项显示出来：

```
sub search_members
{
    my ($dbh, $interest) = @_;
```

```

print p ("Search results for keyword: " . escapeHTML ($interest));
my $sth = $dbh->prepare (qq{
    SELECT * FROM member WHERE interests LIKE ?
    ORDER BY last_name, first_name
});
# look for string anywhere in interest field
$sth->execute ("% " . $interest . "%");
my $count = 0;
while (my $ref = $sth->fetchrow_hashref ())
{
    format_html_entry ($ref);
    ++$count;
}
print p ("Number of matching entries: $count");
}

```

在运行ushl\_browse.pl脚本的时候, 你将发现每次提交的关键字都会出现在下一个页面中的表单里, 即使脚本在生成这个表单的时候把keyword字段的值设定为一个空字符串, 也仍会出现这一现象。造成这一现象的原因是: 如果在脚本的执行环境里存在着某个字段的值, CGI.pm模块就会在生成新表单的时候把这个值自动地填写到那个字段里去。如果你不喜欢这种行为并想让这个字段每次都显示为空白, 就需要给textfield()调用增加一个override参数, 如下所示:

```

print textfield (-name => "keyword",
                 -value => "",
                 -override => 1,
                 -size => 40);

```

search\_members()函数使用了一个辅助函数format\_html\_entry()来显示各个记录项。这个函数与我们为前面内容里的gen\_dir.pl脚本而编写的同名函数大同小异。(请参见第7.3.1节。)不过, 这个函数供gen\_dir.pl脚本使用的那个版本在生成HTML内容时采用的是直接打印HTML排版标记的做法, 而供ushl\_browse.pl脚本使用的这个版本却是通过CGI.pm函数来生成HTML排版标记的:

```

sub format_html_entry
{
    my $entry_ref = shift;

    # encode characters that are special in HTML
    foreach my $key (keys (%{$entry_ref}))
    {
        next unless defined ($entry_ref->{$key});
        $entry_ref->{$key} = escapeHTML ($entry_ref->{$key});
    }
    print strong ("Name: " . format_name ($entry_ref)) . br ();
    my $address = "";
    $address .= $entry_ref->{street}
                if defined ($entry_ref->{street});
    $address .= ", " . $entry_ref->{city}

```



```

        if defined ($entry_ref->{city});
$address .= ", " . $entry_ref->{state}
        if defined ($entry_ref->{state});
$address .= " " . $entry_ref->{zip}
        if defined ($entry_ref->{zip});
print "Address: $address" . br ()
        if $address ne "";
print "Telephone: $entry_ref->{phone}" . br ()
        if defined ($entry_ref->{phone});
print "Email: $entry_ref->{email}" . br ()
        if defined ($entry_ref->{email});
print "Interests: $entry_ref->{interests}" . br ()
        if defined ($entry_ref->{interests});
print br ();
}

```

format\_html\_entry()函数使用了format\_name()函数来把first\_name、last\_name和suffix数据列的取值合并为一个完整的会员姓名。它与我们为前面内容里的gen\_dir.pl脚本而编写的同名函数完全一样。

## 2. 利用FULLTEXT索引来进行搜索

“美国历史研究会”的会员们往往会对多个历史事件感兴趣。在member数据表的interests数据列里，会员们的多个兴趣点（如果有的话）将以逗号分隔，就像下面这样：

```
Revolutionary War,Spanish-American War,Colonial period,Gold rush,Lincoln
```

那么，我们可不可以用ushl\_browse.pl脚本去进行要求同时匹配多个兴趣点的会员记录搜索工作呢？这个问题的答案是：可以，但非常有局限性。ushl\_browse.pl脚本允许在搜索表单里输入多个关键字，但要想找到能同时匹配多个关键字的会员记录，就必须同时满足两个条件：1）必须使用逗号来分隔这些关键字；2）会员记录中的interests数据列里的多个兴趣点与所输入的关键字在排列顺序上完全一致。考虑到这一问题，使用FULLTEXT索引<sup>①</sup>来完成会员兴趣搜索任务就很有必要了，这是一种更为灵活和适用的解决方案。

使用ushl\_ft\_browse.pl脚本的先决条件有两个：1）MySQL软件必须是3.23.23或更高的版本；2）member数据表必须是一个MyISAM数据表。如果当初把member数据表创建成了其他的数据表类型，可以先用下面这条ALTER TABLE语句把它转换为MyISAM数据表：

```
mysql> ALTER TABLE member TYPE = MYISAM;
```

再用下面这条语句在member数据表的interests数据列上创建一个FULLTEXT索引：

```
mysql> ALTER TABLE member ADD FULLTEXT (interests);
```

只有这样，才能用interests数据列进行FULLTEXT搜索。sampdb发行版本里的ushl\_ft\_browse.pl脚本是在前面ushl\_browse.pl脚本的基础上编写出来的，二者只在负责构造检索命令的search\_members()函数上有区别。下面是供ushl\_ft\_browse.pl脚本使用的search\_members()函数的

① 本书的第3章对MySQL的FULLTEXT功能做了比较详细的介绍。

代码:

```
sub search_members
{
    my ($dbh, $interest) = @_;

    print p ("Search results for keyword: " . escapeHTML ($interest));
    my $sth = $dbh->prepare (qq{
        SELECT * FROM member WHERE MATCH(interests) AGAINST(?)
        ORDER BY last_name, first_name
    });
    # look for string anywhere in interest field
    $sth->execute ($interest);
    my $count = 0;
    while (my $ref = $sth->fetchrow_hashref ())
    {
        format_html_entry ($ref);
        ++$count;
    }
    print p ("Number of matching entries: $count");
}
```

这个版本的search\_members()函数对上一节里的同名函数做了以下几处修改:

- 查询命令使用的是MATCH()...AGAINST()子句而不是LIKE子句。
- 没有使用在关键字字符串的两端加上“%”通配符的办法来把它转换为一个匹配模式。

完成上述改动之后,就可以通过Web浏览器去调用ushl\_ft\_browse.pl脚本并在搜索表单里输入多个关键字了(加不加逗号分隔符都不要紧)。这个脚本将把与所输入的关键字中的任何一个相匹配的会员记录查找出来。

这个脚本有很多地方值得改进。比如说,MySQL的FULLTEXT搜索允许同时对多个数据列进行搜索。只需先创建一个同时涵盖多个数据列的FULLTEXT索引,再对ushl\_ft\_browse.pl脚本做相应的修改就能享受到这个好处。下面的第一条命令将丢弃我们现在使用的FULLTEXT索引,而第二条命令将创建一个同时涵盖interests、last\_name和first\_name三个数据列的FULLTEXT索引:

```
mysql> ALTER TABLE member DROP INDEX interests;
mysql> ALTER TABLE member ADD FULLTEXT (interests,last_name,first_name);
```

接着,再把search\_members()函数里的SELECT语句从原来的MATCH (interests)修改为MATCH (interests, last\_name, first\_name),就可以使用新的FULLTEXT索引同时对这三个数据列进行搜索了。

还可以对ushl\_ft\_browse.pl脚本做出另一项改进:在搜索表单里增加两个供脚本使用者选择的单选按钮——Match Any Keyword (匹配任何一个关键字)和Match All Keywords (匹配全部关键字)。Match Any Keyword模式就是这个脚本现在使用的模式。Match All Keywords模式也很容易实现:把查询命令修改为使用一个IN BOOLEAN MODE类型的FULLTEXT搜索,再在每一个关键字的前面加上一个加号(+)字符以表明它必须在搜索过程中得到匹配。有关这方面问题的详细讨论见第3章。

## 第8章 MySQL应用程序设计接口：PHP语言

PHP是一种非常适合编写动态Web网页的脚本语言，用它编写出来的代码能够方便地嵌入到Web页面里。当这个Web页面被访问时，嵌入在其中的PHP代码就会被执行并生成动态的HTML内容，而这些内容将作为Web页面的一部分被送往客户的Web浏览器去显示。这一章的主要内容就是介绍如何编写基于Web的PHP脚本来访问MySQL数据库。在本书的第5章里，我们对PHP语言、C语言、Perl DBI模块等几种MySQL应用程序设计接口（API）进行了比较。

在本书的第1章里，我们创建了一个样板数据库sampdb，并在其中为考试记分项目和“美国历史研究会”分别创建了一些数据表。在这一章里，将以这个样板数据库里的数据表为例来展开讨论。这一章里的脚本都能够运行在PHP 3或PHP 4环境里，但出于性能和安全方面的考虑，推荐使用PHP 4。

这一章里的PHP脚本都是在将与Apache Web服务器配合使用的假设前提下编写的。另外，为了让PHP知道如何去访问MySQL数据库，还必须在建立（编译、链接和安装）PHP的过程中把MySQL C客户程序开发库也链接上。如果你现在还没有安装好上面提到的这几个软件，请参阅附录A。获得本章各示例脚本代码的办法也在可以在附录A里查到——它们是sampdb发行版本的组成部分之一，如果下载了sampdb发行版本，就用不着自己动手输入示例脚本的代码了。与本章有关的脚本都放在sampdb发行版本的phpapi目录里。

在UNIX系统上，PHP既可以被当做Apache服务器的一个模块来使用，也可以像传统的CGI程序那样被当做一个能够独立运行的解释器来使用。在Windows系统上，PHP只能被当做一个独立运行的程序来使用——除非使用的是Apache 2.x和PHP 4。如果是后一种情况，可以选择把PHP运行为Apache服务器的一个模块。在其他的平台上，应该尽可能地把PHP运行为一个模块，因为这有助于获得更好的性能。

限于篇幅，这里只对本章内容涉及到的那些PHP函数做简单的解释，在本书的附录H里，可以查到一份与MySQL有关的PHP函数的完整清单。此外，PHP使用手册也是一份极有参考价值的资料，它对PHP语言中的各种函数（包括那些用来访问其他数据库（PHP并不像第7章里使用的DBI那样只能用来配合MySQL数据库使用）的函数在内）都做了详细的介绍。这个使用手册可以在PHP的官方网站（网址是<http://www.php.net/>）上找到。由Greant等人合著的*PHP Functions Essential Reference*（PHP函数大全，由New Riders公司2002年出版）也是一本非常好的参考书。

PHP脚本的文件名通常都有一个统一的扩展名，因为Web服务器必须根据文件扩展名来识别PHP脚本并调用PHP解释器来执行它们。如果所使用的扩展名不能让Web服务器把文件识别为一个PHP脚本，Web服务器就会把PHP脚本的内容当做普通文本传送出去。这一章里的脚本文件将统一使用.php作为其扩展名。PHP脚本文件的另一种比较常见的扩展名是.phtml。配置Apache服务器以使之能够识别出PHP脚本文件所使用的扩展名的具体步骤可以在附录A中查到。

(如果你没有配置Apache服务器的权限, 请向系统管理员询问你系统上的PHP脚本文件应该使用什么样的扩展名。)还可以让Apache服务器把任何一个名为index.php的脚本用做该脚本所在的目录的默认主页——就像Apache服务器对待index.html文件那样, 有关的配置步骤也可以在附录A里查到。

编写PHP脚本的目的是为了能够使用它们。如果想试用这一章里编写的PHP脚本, 就必须把它们安装到Web服务器能够找到它们的地方去。在这一章里, 所采用的办法是把我们为“美国历史研究会”项目和考试记分项目开发的各有关脚本分别放到Apache文档树顶级目录下为它们专门准备的目录ushl和gp里去。如果想把你的Web服务器也设置成这个样子, 那么现在就应该创建这两个目录。假设Web站点的主机名是www.snake.net, 这两个目录里的页面将有如下所示的URL地址:

```
http://www.snake.net/ushl/...  
http://www.snake.net/gp/...
```

比如说, 如果给这两个目录里的主页起名为index.php, 就可以像下面这样去访问它们:

```
http://www.snake.net/ushl/index.php  
http://www.snake.net/gp/index.php
```

如果已经把Apache服务器配置成使用index.php脚本作为各有关目录的默认主页的话, 下面两个URL地址与上面两个URL地址将是等价的:

```
http://www.snake.net/ushl/  
http://www.snake.net/gp/
```

## 8.1 PHP语言概述

PHP的基本用途是以解释方式执行脚本以生成一个将被送往某个客户的Web页面, 而脚本通常是一个夹杂着HTML文本与可执行PHP代码的混合体。PHP脚本中的HTML文本将按原样发送给客户, 而其中的PHP代码却会在发送给客户的Web页面里被替换为执行这段代码所生成的输出内容。也就是说, 客户不会看到PHP脚本中的可执行代码, 他们只能看到这些代码所生成的HTML页面<sup>①</sup>。

当Web服务器需要读取一个PHP脚本文件的时候, 它会调用PHP解释器来完成这个读操作(如果已经把Web服务器配置好了的话)。PHP解释器会把它遇到的任何东西简单地拷贝到输出(就好像这个文件的内容完全是普通的HTML文本那样), 直到它遇到一个特殊的起始标记; 一旦遇到特殊的起始标记, PHP解释器就将从HTML模式切换到PHP代码执行模式并从那里开始把该文件的后续内容解释为可执行PHP代码直到它遇到一个特殊的结束标记; 一旦遇到特殊的结束标记, PHP解释器就将从PHP代码执行模式切换回HTML模式。这意味着可以把静态文本(PHP脚本中的HTML部分)与动态文本(PHP脚本中的PHP代码部分的执行结果)混合生成一个会随着这个脚本的调用环境而发生变化的Web页面。比如说, 可以让PHP脚本在Web页面里生

<sup>①</sup> 这一章里的PHP脚本所生成的页面都符合XHTML规范, 而不是仅符合HTML规范。第7.4.2节对XHTML做了简单的介绍。

成一个表单供用户输入数据库搜索参数。如果用户在表单里输入的搜索参数不同，脚本根据这个参数而从数据库里检索出来的信息也就不同，于是脚本生成并返回给用户的搜索结果页面也就不同。

我们就用下面这个最简单的例子里来开始PHP学习之旅：

```
<html>
<body>
<p>hello, world</p>
</body>
</html>
```

这个脚本实在是太简单了——其中根本就没有PHP代码！我想你们肯定会问：“这么简单的脚本有什么用？”我理解大家的心情，但事情是这样的：在编写PHP脚本的时候，先搭建出它将要生成的Web页面的HTML框架再往里面填充PHP代码的做法往往是一个很不错的思路。而且，这种做法是绝对允许的，PHP解释器也绝对不会有抱怨。

如果想把PHP代码添加到一个脚本里去，就必须把它们放在特殊的起始标记“<?php”和结束标记“?”之间以区别于周围的HTML文本。在遇到起始标记“<?php”的时候，PHP解释器会从HTML模式切换到PHP代码执行模式并从那里开始把该文件的后续内容解释为可执行PHP代码直到它遇到结束标记“?”。这两个标记之间的代码将在最终生成的结果页面里被替换为PHP解释器以解释方式执行这些代码而产生的输出内容。现在，我们来修改一下上面那个例子，给它加上一小段PHP代码，如下所示：

```
<html>
<body>
<p><?php print ("hello, world"); ?></p>
</body>
</html>
```

这个脚本里的PHP代码部分算得上是最小的了，只有一行代码而已。这行PHP代码的执行结果是打印出“hello, world”，这个结果将作为整个脚本的输出内容的一部分被送往客户的浏览器。换句话说，这个脚本所生成的Web页面与前面那个完全由HTML文本构成的脚本所生成的Web页面是一模一样的。

PHP代码可以用来生成Web页面的任何一个部分。一个极端是整个脚本完全由HTML文本构成而不包含任何PHP代码；另一个极端则是全部HTML内容都由PHP代码来生成，如下所示：

```
<?php
print("<html>\n");
print("<body>\n");
print("<p>hello, world</p>\n");
print("</body>\n");
print("</html>\n");
?>
```

这三个例子证明了这样一件事：PHP能够让我们以非常灵活的方式去生成Web输出，脚本中的HTML文本与PHP代码的“混合比例”完全由我们自己来控制。PHP的灵活性还表现在它



并不要求PHP代码全都出现在脚本里的同一个地方,它允许把PHP代码散布在脚本里的任意位置——你可以在HTML模式与PHP代码执行模式之间随意切换,想切换多少次都行。

除上面这几个例子里使用的“<?php”和“?>”以外,PHP还允许使用其他风格的标记。PHP支持使用的各种标记以及它们的用法可以在附录H里查到。

### 可独立执行的PHP脚本

这一章里的示例脚本都是以供Web服务器调用来生成Web页面为目标而编写出来的。不过,如果你手里还有一个可独立运行的PHP版本,就可以用它在命令行上来执行PHP脚本。比如说,假设有一个内容如下所示的hello.php脚本:

```
<?php print ("hello, world\n"); ?>
```

那么,就可以用下面这条命令亲自在命令行上执行它:

```
% php -q hello.php
hello, world
```

这对PHP脚本的开发工作是有好处的,因为它能使你立刻发现脚本是否存在语法错误或其他问题而不必每修改一次脚本都不得不通过Web浏览器去查看它的执行情况。出于这个考虑,即使你们通常都把PHP用做Apache服务器的一个模块,也建议大家在自己的系统上再建立一个可独立运行的PHP版本。

如果我们在PHP脚本的开头加上一个用来给出PHP路径名的“#!”行(就像shell或Perl脚本里那样),就可以让它变成一个可以在命令行上直接执行的文件。假设PHP解释器安装在目录/usr/local/bin里,可以把上面的脚本修改为如下所示的样子:

```
#!/usr/local/bin/php -q
<?php print ("hello, world\n"); ?>
```

再用“chmod +x”命令把它设置为可执行文件,就可以像下面这样来调用它了:

```
% chmod +x hello.php
% ./hello.php
hello, world
```

如果PHP的用途只不过是打印语句来生成一些用静态HTML文本也能实现的Web页面的话,它也就没多大用处了。PHP的真正威力是它能够让脚本根据不同的调用环境去动态地生成不同的Web输出。下面这个脚本就演示了这一点。虽然它仍比较短小,但比前面那几个例子却多了一些内涵。它让我们看到从PHP访问MySQL数据库和把数据库查询结果用在一个Web页面里是多么容易。这个脚本在第5章里曾经介绍过,它给出了“美国历史研究会”主页的一个简单框架。随着学习的深入,我们将逐步地使这个脚本得到进一步的充实和完善,但目前它仅能显示一个简短的欢迎信息和显示现有会员总人数:

```
<html>
<head>
<title>U.S. Historical League</title>
</head>
```

```

<body bgcolor="white">
<p>Welcome to the U.S. Historical League Web Site.</p>
<?php
# USHL home page

$conn_id = @mysql_connect ("cobra.snake.net", "sampadm", "secret")
    or exit ();
mysql_select_db ("sampdb")
    or exit ();
$result_id = mysql_query ("SELECT COUNT(*) FROM member")
    or exit ();
if ($row = mysql_fetch_row ($result_id))
    print ("<p>The League currently has " . $row[0] . " members.</p>");
mysql_free_result ($result_id);
?>
</body>
</html>

```

欢迎信息是静态文字内容，所以用静态HTML文本就很容易实现出来。现有会员总人数却是一项动态内容，会随着时间的推移而发生变化，只能通过对sampdb数据库里的member数据表进行一次即时查询的办法来确定。

这个脚本的代码部分完成了以下几项简单的工作：

- 1) 连接MySQL服务器，并把sampdb数据库设置为当前的默认数据库。
- 2) 向MySQL服务器发送了一个用来确定“美国历史研究会”现有会员总人数的查询（具体做法是统计member数据表的数据行总数）。
- 3) 根据查询结果生成了一条报告会员总人数的消息并释放了这个查询结果集。

在上述过程中，只要执行出错，脚本就会退出执行而不产生任何进一步的输出。当因执行出错而退出执行的时候，为了不使那些到访这个站点的人感到困惑，这个脚本将不显示出错信息<sup>①</sup>。

可以在sampdb发行版本的phpapi/ushl目录里找到这个脚本，它的文件名是index.php。在对有关的数据库连接参数做了必要的修改并把这个index.php文件拷贝到Web服务器文档树的ushl目录里之后，就可以通过下面两个URL地址中的任何一个来访问这个脚本了：

```

http://www.snake.net/ushl/
http://www.snake.net/ushl/index.php

```

下面，我们对这个脚本中的有关代码做一下分解说明。第一步是通过一个mysql\_connect()调用来连接MySQL服务器：

```

$conn_id = @mysql_connect ("cobra.snake.net", "sampadm", "secret")
    or exit ();

```

① 不过，如果你的某个Web页面要完全依靠PHP代码来生成的话，因脚本执行出错而退出却不给出任何输出信息会让那些到访站点的人们感到困惑和不便——因为有些浏览器会在遇到这种情况时在客户的浏览器画面里显示一个“This page contained no data”（本页面不包含数据）对话框，对方在困惑之余还得再点击某个按钮才能消除这个对话框。对于这类情况，最好显示一个页面来告诉对方他们的请求暂时无法得到满足。

这个函数需要给出三个输入参数,依次是MySQL服务器主机的主机名、MySQL账户的用户名和该账户的口令。如果连接成功,mysql\_connect()函数将返回一个连接标识符;如果执行出错,则返回FALSE。如果连接失败,脚本将立刻调用exit()函数终止执行。

不过,像上面这样把MySQL账户的用户名和口令嵌在脚本里多少有点不安全。在正常情况下,它们是不会出现在这个脚本所生成的Web页面里的——因为脚本内容将被脚本代码的执行输出所替代。可万一Web服务器因配置不当而没能识别出这个脚本需要它调用PHP解释器来处理,它就会把脚本代码当做普通文本发送出去,你的数据库连接参数就将暴露在别人面前。这个问题留到第8.1.1节里去解决。

出现在mysql\_connect()调用前面的那个“@”字符是干什么的?这个字符是“请闭嘴”操作符。有些PHP函数在调用失败的时候不仅会返回一个状态代码,还会写出一条出错信息。以mysql\_connect()函数为例,如果它没能建立与MySQL服务器的连接,就会导致一条下面这样的出错信息出现在将被发送到客户端浏览器去的Web页面里:

```
Warning: MySQL Connection Failed: Access denied for user:
'sampadm@cobra.snake.net' (Using password: YES)
```

很明显,这类出错信息没必要让来到站点的那些访问者看到,他们要么看不懂这条信息,要么不知道该怎么办。我们也不需要这条信息——完全可以根据这个函数调用的返回值来决定应该怎样去处理这类错误。把“@”字符放在mysql\_connect()调用的前面就能使它不把这类出错信息写出来。以这个脚本为例,当真的出现这种情况的时候,最好的办法是根本就不生成与会员总人数有关的任何输出——就让那个页面只包含一条欢迎信息好了。

完全可以把“@”操作符放在任何一个PHP表达式的前面。但根据经验,最容易发生问题的就是第一个调用,因此,在本章的各有关示例脚本里,只在mysql\_connect()函数调用的前面放上一个“@”操作符以消除它可能产生的出错信息。(如果有必要进行其他的出错处理的话,让脚本打印一条它自己的出错信息。)

#### mysql\_connect() 与 mysql\_pconnect()

PHP语言里还有一个与mysql\_connect()函数功能相近的函数叫mysql\_pconnect()。它们都以主机名、用户名和口令为输入参数,在调用成功时都会返回一个连接标识符,失败时也都返回FALSE。这两个调用的区别是:mysql\_connect()建立的是一个非永久性连接,而mysql\_pconnect()建立的却是一个永久性的连接。永久性连接与非永久性连接的区别是前者不会在脚本终止时被关闭,如果同一进程稍后执行的其他PHP脚本里还有一个参数完全相同的mysql\_pconnect()调用,它就会直接使用这个永久性连接而不需要再发出mysql\_pconnect()调用了。对很多数据库引擎来讲,使用永久性连接的做法要比每次都去建立和关闭一个非永久性连接的做法更有效。但这对MySQL并没有多大的好处,因为针对MySQL数据库的连接建立工作已经非常有效了。事实上,在MySQL数据库上使用永久性连接说不定还会弄巧成拙——如果Web服务器非常繁忙并需要调用大量的PHP脚本,让Apache保持过多的永久性MySQL连接就有可能迅速耗尽系统的可用连接资源。虽说加大服务器变量max\_connections的值能够在一定程度上缓解这一问题(其具体做法见第11.5.6节),但使用非永久性连接往往是一个更好的选择。

mysql\_connect()调用所返回的连接标识符可以传递到PHP API的其他几个与MySQL有关的函数调用里去。不过,那些函数调用里的连接标识符参数往往是可选的,比如说,mysql\_select\_db()函数的下面两种调用方式都是允许的:

```
mysql_select_db ($db_name, $conn_id);
mysql_select_db ($db_name);
```

在与MySQL有关的PHP函数里,有几个是需要连接标识符参数的,但如果在调用它们的时候没有给出这个参数,它们就将默认地使用最近打开的那个连接。换句话说,如果脚本只打开了一个连接,有关函数就将默认地使用这个连接,而不必在调用这些函数的时候明确地指定一个连接参数了。这与使用C语言或Perl语言API来开发MySQL应用程序时的做法是有区别的——C语言或Perl语言API里没有这样的默认设置。

为了让大家看清楚mysql\_pconnect()调用的返回值类型,我们这个简单的主页脚本里的连接代码特意使用了\$conn\_id变量,如下所示:

```
$conn_id = @mysql_connect ("cobra.snake.net", "sampadm", "secret")
or exit ();
```

请注意,这个\$conn\_id变量并没有在脚本的其他地方出现过,所以这条语句其实可以更简单地写成下面这个样子:

```
@mysql_connect ("cobra.snake.net", "sampadm", "secret")
or exit ();
```

如果成功地建立起与MySQL服务器的连接,下一步就是选取一个数据库:

```
mysql_select_db ("sampdb")
or exit ();
```

如果mysql\_select\_db()调用失败,脚本将默默地退出执行。一般说来,如果已经成功地连接上了MySQL而所选取的数据库又的确存在,这个调用是不太容易失败的;但为保险起见,还是应该对这个问题有所准备并做出适当的处理。

在选取了数据库之后,脚本向MySQL服务器发去一个要求统计会员总人数的查询,提取出查询结果,显示这个结果,然后释放了查询结果集:

```
$result_id = mysql_query ("SELECT COUNT(*) FROM member")
or exit ();
if ($row = mysql_fetch_row ($result_id))
    print ("<p>The League currently has " . $row[0] . " members.</p>");
mysql_free_result ($result_id);
```

mysql\_query()函数负责把SQL命令发送给MySQL服务器去执行。(注意:查询命令字符串的末尾不需要使用分号字符“;”或字符序列“\g”来终止,这与我们在mysql程序里进行查询时的做法是不一样的。)如果查询命令有语法错误或者因为某种原因而无法执行,mysql\_query()函数将返回FALSE;否则,它将返回一个结果集标识符。这个标识符是一个我们可以用来获得关于该结果集的各种信息的值。就这个脚本所进行的查询来说,它的结果集仅有一个数据行,而这个数据行又仅有一个数据列,即一个代表会员总人数的计数值。为了得到这个计数值,我们



把结果集标识符传递到了mysql\_fetch\_row()函数里以取回其中的数据行,把这个数据行赋值给变量\$row,然后访问它的第一个(恰好也是它惟一的一个)元素\$row[0]。

完成了对结果集的处理之后,我们再把它的结果集标识符传递到mysql\_free\_result()函数里以释放它。在这里,这个调用其实并不必要,因为PHP会在脚本结束执行时自动释放所有尚未被释放的结果集,把它写在这个脚本里的目的是为了让大家对使用PHP脚本来访问MySQL数据库的流程有一个完整的认识。mysql\_free\_result()函数主要用在脚本需要对MySQL数据库进行大规模查询或者进行大量查询的场合,它有助于防止脚本占用过多的内存。

### PHP脚本中的变量

PHP脚本里的变量不需要事先做出声明或者定义,开始使用某个变量的事实就是它得以存在的理由。我们的主页脚本使用了\$conn\_id、\$result\_id和\$row三个变量,但在使用前都没有对它们做出声明。(有些上下文需要对变量做出明确的声明,比如当需要在某个函数里引用一个全局变量的时候,稍后会讲到这个问题。)

不管将被用来表示哪种类型的值,PHP变量的名字都必须是以一个美元字符(\$)打头的标识符。但如果需要通过数组或者对象的变量名来访问该变量值里的各个元素,那就还得再额外增加点东西。具体地说,如果变量\$x表示的是一个标量值(比如一个数字或者一个字符串),就可以直接通过变量名\$x来访问;如果变量\$x表示的是一个以数字为下标的数组,就需要通过\$x[0]、\$x[1]等构造来访问其中的各个元素;如果变量\$x表示的是一个以"yellow"或"large"之类的字符串为下标的数组(即所谓的关联数组),就需要通过\$x["yellow"]或\$x["large"]之类的构造来访问其中的各个元素(PHP数组甚至允许在同一个数组里混合使用数字和字符串下标。比如说,\$x[1]和\$x["large"]允许是同一个数组里的元素);如果变量\$x表示的是一个对象,就需要通过\$x->property\_name之类的构造(比如\$x->yellow和\$x->large等)来访问该对象的属性。注意:PHP不允许以数字作为对象的属性名,所以像\$x->1这样的构造在PHP里是非法的。

### PHP语言上的影响

如果你有C语言程序设计经验,就应该注意到PHP脚本里有很多构造与在编写C语言程序时使用的非常相似。PHP的大部分语法都沿袭自C语言,所以这两种程序设计语言之间的相似性绝非巧合。在C语言上的编程经验几乎能全部运用到PHP脚本的设计工作中。事实上,如果拿不准PHP脚本里的某个表达式或者某个控制结构应该如何编写,不妨试试C语言程序里的做法——它们几乎总是能解决问题。

虽然PHP沿袭了C语言的很多东西,但Java和Perl语言对它的影响也随处可见。这一点在PHP的注释语法里表现得尤其明显。下面这几种注释形式在PHP脚本里都是允许的:

- # Perl语言风格的注释;从“#”开始直到这一行的行尾。
- // C++或Java语言风格的注释;从“//”开始直到这一行的行尾。
- /\* C语言风格的注释;从“/\*”开始,到“\*/”结束。



PHP与Perl的其他相似之处还包括：字符串合并操作符都使用的是句点“.”（字符串追加合并操作符也都使用的是“.=”）；变量引用和转义序列都只能在双引号里才能得到解释，在单引号里则都不能得到解释；等等。

### 8.1.1 函数与include文件的使用

PHP脚本与DBI脚本的主要区别之一是PHP脚本必须被放到Web服务器的文档树里才能使用，而用来存放DBI脚本的场所（以cgi-bin目录最为典型）却很少（如果还有的话）落在Web服务器的文档树里。这就引发了一个安全方面的问题：配置不当的Web服务器很可能会把Web文档树里的脚本程序当做普通文本发送到客户那里去。这意味着把用来连接MySQL服务器的用户名和口令嵌在PHP脚本里的做法比把它们嵌在DBI脚本里的做法要冒更大的泄密风险。

前面编写的“美国历史研究会”主页脚本就存在着这一问题，因为其中的确有用来连接MySQL服务器的用户名和口令。下面，我们将利用PHP所提供的两种机制（函数和include文件）来把这些数据库连接参数转移到脚本的外部去：我们将编写一个名为sampdb\_connect()的函数并把它放到一个include文件里去，这个include文件与主脚本分属两个文件，但可以从脚本代码里引用它。这种做法的好处主要有以下几点：

- 用来连接MySQL服务器的代码更容易编写和修改。连接参数只需在辅助函数sampdb\_connect()里写出一次，而不是必须在每一个有关的脚本里都得写出一次。此外，因为我们将要编写的脚本都要与sampdb数据库打交道，所以还可以把“连接MySQL服务器”这一步骤之后的“选取当前数据库”步骤加到这个辅助函数里去，即让它负责完成两个MySQL操作。这种隐藏操作细节的做法不仅能够改善脚本的可读性，而且还能让你把注意力更多地集中到各脚本所要完成的任务上去，因为你不需要再去操心数据库连接方面的问题了。
- 一个include文件可以用于多个脚本。这一方面能提高代码的可复用性，另一方面能使代码的维护工作更容易进行。在include文件里所做的改动将反映在使用了这个include文件的每一个脚本里。比如说，如果今后需要把sampdb数据库从主机cobra转移到主机boa上，用不着一个一个地去修改一大堆的脚本，只要在对sampdb\_connect()函数做出定义的那个include文件里把mysql\_query()调用中的主机名参数改过来就行了。
- include文件可以不放在Apache服务器的文档树里。这意味着客户们将不能通过他们的Web浏览器去直接请求include文件，即便是Web服务器配置不当，include文件里的内容也不会暴露给客户。如果有一些不想被Web服务器发送出站点的敏感信息，用include文件来隐藏它们的策略很值得考虑。不过，虽然增加了一些安全性，但这种做法并不能百分之百地保证MySQL用户名和口令不会被泄露。如果没有进一步采取一些预防措施，那些在Web服务器主机上有登录账户的其他用户将仍有可能直接读出include文件的内容。在第7.4.3节里给出了一些与DBI配置文件的安装工作有关的建议，那些建议同样适用于PHPinclude文件的安装工作。

在开始使用include文件之前,需要先安排一个地方来存放它们,还得把这个地方告诉PHP好让它能找到它们。如果你在自己的系统上已经安排了一个这样的地方,可以直接使用之;如果还没有这样做,请参照以下步骤来为include文件安排去处:

1) 在Web服务器文档树以外的某个地方创建一个专门用来存放PHPinclude文件的目录。这里为PHPinclude文件而创建的目录是/usr/local/apache/lib/php,它位于Web服务器文档树(/usr/local/apache/htdocs)以外,没有放在其中。

2) 在脚本里,可以通过include文件的完整路径名来访问它;或者,如果已经设置好PHP搜索路径的话,也可以直接使用它们的文件名(即它们完整路径名的最后一部分)<sup>①</sup>。建议大家采用设置PHP搜索路径的做法,让PHP替你去寻找include文件能省不少事。PHP用来寻找include文件的搜索路径由PHP初始化文件php.ini里的配置参数include\_path控制。在系统上找到这个文件(我把这个文件安装在系统上的/usr/local/lib目录里),再找到以include\_path开始的那一行。如果它还没有设置值,请把刚才为存放include文件而新创建的目录的完整路径名写在那里,如下所示:

```
include_path = "/usr/local/apache/lib/php"
```

如果include\_path已经有了一个设置值,就需要把新路径追加到它的尾部,如下所示:

```
include_path = "current_value:/usr/local/apache/lib/php"
```

在UNIX系统上,列在include\_path里的目录必须以冒号(:)来分隔,就像上面那样;在Windows系统上,它们必须以分号(;)来分隔。

3) 创建include文件并把它放到用来存放include文件的目录里去。注意,include文件的名字不得重复,这里使用sampdb.php。这个文件最终将包含几个函数,但作为开端,目前其中仅有sampdb\_connect()这一个函数。下面是这个函数的代码:

```
<?php
# sampdb.php - sampdb sample database common functions

# Connect to the MySQL server using our top-secret name and password

function sampdb_connect ()
{
    $conn_id = @mysql_connect ("cobra.snake.net", "sampadm", "secret");
    if ($conn_id && mysql_select_db ("sampdb"))
        return ($conn_id);
    return (FALSE);
}
?>
```

如果sampdb\_connect()函数成功地连接上MySQL服务器并选中了sampdb数据库,它将返回一个连接标识符;如果执行出错,它将返回FALSE。sampdb\_connect()函数在执行出错时不会打

① PHP include文件的使用方法与C语言中的include文件很相似。比如说,C语言的预编译处理器会到多个目录里去寻找C程序的include文件,而PHP也是这么做的。

印出任何消息，可以根据各有关脚本的具体任务来决定这个函数的调用者（即你的脚本）是默默地退出执行还是打印一些信息。

注意，sampdb.php文件里的PHP代码要用脚本标记“<?php”和“?>”括起来。这是因为PHP将在HTML模式里开始读取include文件的内容。如果忽略了这对标记，PHP就会把include文件的内容当做普通文本发送出去而不会把它们解释为PHP代码。（如果include文件就是用来按原样输出HTML文本的，这么做当然没问题；可如果想让它的内容得到执行，就必须把PHP代码用脚本标记括起来。）

4) 在脚本里，如果需要引用某个include文件，请在引用之前写出一条下面这样的语句：

```
include "sampdb.php";
```

在看到这条语句的时候，PHP会去搜索这个include文件并读入它的内容，此后的脚本代码就能随意访问这个include文件里的任何东西了。

在sampdb发行版本中的phpapi目录里有一个现成的sampdb.php文件，把它拷贝到用来存放include文件的目录里，再把它的主和访问模式设置为允许Web服务器读取。如果想利用这个include文件来连接MySQL服务器，可能还需要对连接参数做些相应的修改。

安装好sampdb.php文件之后，我们就可以让“美国历史研究会”的主页脚本通过引用这个include文件里的sampdb\_connect()函数来连接MySQL服务器了，如下所示：

```
<html>
<head>
<title>U.S. Historical League</title>
</head>
<body bgcolor="white">
<p>Welcome to the U.S. Historical League Web Site.</p>
<?php
# USHL home page - version 2

include "sampdb.php";

sampdb_connect ()
    or exit ();
$result_id = mysql_query ("SELECT COUNT(*) FROM member")
    or exit ();
if ($row = mysql_fetch_row ($result_id))
    print ("<p>The League currently has " . $row[0] . " members.</p>");
mysql_free_result ($result_id);
?>
</body>
</html>
```

这个脚本可以在sampdb发行版本的phpapi/ushl目录里找到，它的文件名是index2.php。把它拷贝到Web服务器文档树的ushl目录并改名为index.php，即用它来替换掉我们刚才放在那个目录里的同名脚本。因为用来连接MySQL服务器的用户名和口令不再出现于新文件里，所以这一系列操作将把刚才那个不太安全的脚本替换为现在这个比较安全的脚本。

### include语句与require语句

PHP还提供了一个功能与include相近的require语句。它们之间的区别是: PHP只在每次执行到include语句时才读入该语句所给定的文件; 但require语句却会被它所给定的文件的内容替换掉而不管这条语句是否会被执行到。这意味着如果脚本需要多次引用同一个include文件, 使用require语句来引用它们将更有效。反之, 如果脚本每次执行时所引用的include文件都各不相同或者脚本里有一个用来对一组include文件进行遍历的循环, 使用include语句就更方便些——可以安排一个变量来存放include文件的名字, 再把这个变量用做include语句的参数。

PHP 4新增了两个相关的语句——include\_once和require\_once。它们的作用与include和require语句类似, 但如果指定的include文件已经被读入脚本, 这两条新增的语句将不再重复读入之。如果include文件里还包含着另外一些include文件, 使用这两条语句将避免多次读入同一个include文件——多次读入同一个include文件可能导致脚本报告出现函数重复定义错误。

如果你们现在还觉得使用include文件来编写主页脚本并没有减少多大的工作量, 请耐心等待。这个sampdb.phpinclude文件还会在编写其他函数的时候用到, 我们还会把其他一些会在多个脚本里用到的函数也收录到这个include文件里, 使它逐步完善为一个方便的“函数库”。事实上, 现在就可以再往这个include文件里添加两个新的公共函数。先说第一个。这一章里编写的每个Web脚本都将生成一组相当套路化的HTML排版标记来作为各有关主页的开头和结尾部分, 与其在每个脚本里都不得不重复编写一些代码去生成那些几乎完全一样的HTML标记, 不如利用现在这个机会编写两个函数(html\_begin()和html\_end())来让它们替我们生成这些标记。其中, html\_begin()函数的两个输入参数将分别给出有关主页的页面标题(\$title)和文档标题(\$header)。下面是这两个函数的代码:

```
function html_begin ($title, $header)
{
    print("<html>\n");
    print("<head>\n");
    if ($title != "")
        print("<title>$title</title>\n");
    print("</head>\n");
    print("<body bgcolor=\"white\">\n");
    if ($header != "")
        print("<h2>$header</h2>\n");
}

function html_end ()
{
    print("</body></html>\n");
}
```

把html\_begin()和html\_end()函数放到sampdb.phpinclude文件里之后,我们就可以修改“美国历史研究会”的主页脚本来使用它们了。下面是修改后的主页脚本:

```
<?php
# USHL home page - version 3

include "sampdb.php";

$title = "U.S. Historical League";
html_begin ($title, $title);
?>

<p>Welcome to the U.S. Historical League Web Site.</p>

<?php
sampdb_connect ()
    or exit ();
$result_id = mysql_query ("SELECT COUNT(*) FROM member")
    or exit ();
if ($row = mysql_fetch_row ($result_id))
    print ("<p>The League currently has " . $row[0] . " members.</p>");
mysql_free_result ($result_id);

html_end ();
?>
```

请注意,这个脚本里的PHP代码被分隔成了两个部分,两段PHP代码的中间有一条用来显示欢迎信息的HTML文本。

利用include文件里的函数来生成Web页面的开头和结尾部分有一个非常大的好处:如果想统一改变各有关页面里的文档标题和文档脚注,只需对include文件里的相关函数做出修改,那些修改就将自动反映在使用了这些函数的每一个脚本里。比如说,如果想在“美国历史研究会”每一个页面的末尾显示一条版权信息“Copyright USHL”,只需在用来生成各页面结尾部分的html\_end()函数里加上一条打印语句(这很容易做到)就行了,不必再一个一个地去修改各有关脚本。

### 8.1.2 一个简单的数据检索页面

我们嵌在“美国历史研究会”主页脚本里的PHP代码所运行的数据库查询命令只返回了一个数据行(即会员总人数的统计值)。我们的下一个脚本将演示如何对一个由多个数据行构成的结果集(member数据表的全部内容)进行处理。这个PHP脚本与我们在第7章开发的DBI脚本dump\_members.pl在功能上完全一样,所以给它起名为dump\_members.php。DBI脚本通常是在命令行上被调用执行的,而这里的PHP脚本却通常要通过Web服务器来调用执行。因此,PHP脚本的输出通常是一些HTML内容而不是一些以制表符分隔的文本。为了让从数据库里检索出来的数据行和数据列能整齐地显示出来,dump\_members.php脚本将把从member数据表里检索出



来的会员记录写到一个HTML表格里去。下面就是这个脚本的代码:

```
<?php
# dump_members.php - dump Historical League membership list as HTML table

include "sampdb.php";

$title = "U.S. Historical League Member List";
html_begin ($title, $title);

sampdb_connect ()
    or die ("Cannot connect to server");

# issue query
$query = "SELECT last_name, first_name, suffix, email,"
        . "street, city, state, zip, phone FROM member ORDER BY last_name";
$result_id = mysql_query ($query)
    or die ("Cannot execute query");

print("<table>\n");                # begin table
# read results of query, then clean up
while ($row = mysql_fetch_row ($result_id))
{
    print("<tr>\n");                # begin table row
    for ($i = 0; $i < mysql_num_fields ($result_id); $i++)
    {
        # escape any special characters and print table cell
        printf("<td>%s</td>\n", htmlspecialchars ($row[$i]));
    }
    print("</tr>\n");                # end table row
}
mysql_free_result ($result_id);
print("</table>\n");                # end table

html_end ();
?>
```

如果在执行时出现错误, 这个脚本将使用die()函数来打印出错信息并退出执行<sup>①</sup>。这与我们在“美国历史研究会”主页脚本里使用的办法是不同的。前面那个主页脚本的主要功用是显示一条欢迎信息, 统计并打印现有会员总人数的工作只是一项副业, 所以我们不需要它在执行出错时给出一条出错信息。但dump\_members.php脚本的存在价值就在于显示从数据库查询出来的结果, 所以当它因为某种原因而无法显示数据库查询结果时, 就应该给出一条出错信息以表明发生了这样的错误。

如果想试用dump\_members.php脚本, 请先把它安装到Web服务器文档树的ushl目录里, 再

<sup>①</sup> die()函数与exit()很相似, 但它会在退出前先打印一条消息。

通过一个下面这样的URL地址来访问它：

```
http://www.snake.net/ushl/dump_members.php
```

验证无误后，在“美国历史研究会”的主页脚本里增加一个指向脚本dump\_members.php的链接，以便人们能够通过他们的Web浏览器来访问它。下面是增加了这一链接后得到的主页脚本代码：

```
<?php
# USHL home page - version 4

include "sampdb.php";

$title = "U.S. Historical League";
html_begin ($title, $title);
?>

<p>Welcome to the U.S. Historical League Web Site.</p>

<?php
sampdb_connect ()
    or exit ();
$result_id = mysql_query ("SELECT COUNT(*) FROM member")
    or exit ();
if ($row = mysql_fetch_row ($result_id))
    print ("<p>The League currently has " . $row[0] . " members.</p>");
mysql_free_result ($result_id);
?>

<p>
You can view the directory of members <a href="dump_members.php">here</a>.
</p>

<?php
html_end ();
?>
```

编写dump\_members.php脚本的主要目的有两个：一是为了向大家演示如何利用PHP脚本来检索MySQL数据库里的信息，二是为了演示如何把从数据库检索出来的信息生成为Web页面的内容。如果想得到更精细的输出效果，可以自行修改这个脚本。比如说，可以把来自email数据列里的信息显示为一个超链接而不是静态文本，这样，来到“美国历史研究会”站点的访问者只需点击这个超链接就可以向研究会的会员发电子邮件了。sampdb发行版本里的dump\_members2.php脚本能够完成这一任务，它与dump\_members.php脚本只在用来取回和显示会员记录项的循环语句里稍有区别。下面是dump\_members.php脚本里的循环语句：

```
while ($row = mysql_fetch_row ($result_id))
{
    print ("<tr>\n");
    # begin table row
```

```

for ($i = 0; $i < mysql_num_fields ($result_id); $i++)
{
    # escape any special characters and print table cell
    printf("<td>%s</td>\n", htmlspecialchars ($row[$i]));
}
print("</tr>\n");          # end table row
}

```

会员们的电子邮件地址放在数据库查询结果的第4个数据列里。于是,如果这第4个数据列里的值不为空,我们就让dump\_members2.php脚本把它生成为一个超链接,其他数据列的处理情况保持不变。下面是dump\_members2.php脚本里的循环语句:

```

while ($row = mysql_fetch_row ($result_id))
{
    print("<tr>\n");          # begin table row
    for ($i = 0; $i < mysql_num_fields ($result_id); $i++)
    {
        print("<td>");
        if ($i == 3 && $row[$i] != "") # email is in 4th column of result
        {
            printf("<a href='mailto:%s'>%s</a>",
                $row[$i],
                htmlspecialchars ($row[$i]));
        }
        else
        {
            # escape any special characters and print table cell
            print(htmlspecialchars ($row[$i]));
        }
        print("</td>\n");
    }
    print("</tr>\n");          # end table row
}

```

### 8.1.3 对查询结果进行处理

我们将在这一小节对PHP用来执行MySQL查询命令和处理结果集的各种机制做进一步的详细介绍。PHP脚本里的数据库查询命令是通过调用mysql\_query()函数来发出的,这个函数要求提供一个查询命令字符串和一个连接标识符作为它的输入参数。不过,连接标识符参数允许省略,所以下面两种调用mysql\_query()函数的办法都是允许的:

```

$result_id = mysql_query ($query, $conn_id); # use explicit connection
$result_id = mysql_query ($query);          # use default connection

```

mysql\_query()调用将返回一个结果集标识符,这个结果集标识符的含义必须根据脚本所发出的查询命令的类型来解释。对于那些不返回数据行的非SELECT语句,如DELETE、INSERT、REPLACE或UPDATE等语句,mysql\_query()调用将返回TRUE或FALSE以表明查询是否成功。

如果查询成功，还可以调用mysql\_affected\_rows()函数来查看它到底改变（指删除、插入、替换或者修改）了多少个数据行。

对于SELECT语句，mysql\_query()调用将返回一个结果集标识符（查询成功）或FALSE（查询失败）。如果查询成功，就可以利用这个结果集标识符来进一步获取与该结果集有关的各种信息。比如说，如果想知道结果集里有多少个数据行或数据列，可以调用mysql\_num\_rows()或mysql\_num\_fields()函数。如果想访问结果集里的各个记录项，可以在PHP提供的几个数据行取回函数里挑一个来调用。

当mysql\_query()调用返回FALSE的时候，它的含义是数据库查询命令没有执行成功——换句话说，查询命令因为发生了一些错误而根本没有来得及执行。SELECT查询执行失败的原因主要有以下几种：

- 查询命令本身包含语法错误。
- 查询命令虽然语法正确，但语义不正确。比如，试图选取的某个数据列在指定的数据表里根本就不存在等等。
- 没有执行这条查询命令所需要的权限。
- 无法连接MySQL服务器，比如网络出现故障等情况。

如果mysql\_query()调用返回的是FALSE而你还想了解具体的失败原因，可以调用mysql\_error()或mysql\_errno()函数来获得一个出错信息字符串或一个数值形式的出错代码（详见第8.1.5节）。

#### 千万不要想当然地认为mysql\_query()调用肯定会成功

在PHP邮件列表上，经常能看到一些PHP新手问为什么他的脚本打印出了一条下面这样的出错信息：

```
Warning: 0 is not a MySQL result index in file on line n
```

这条消息的含义是：提问者把一个取值为0（即FALSE）的结果集标识符传递给了一些预期着一个有效的结果集标识符的函数（比如某个数据行取回函数）。这意味着此前的mysql\_query()调用返回了FALSE（换句话说，此前的mysql\_query()调用失败了），而那位提问者根本没有对此进行检查就把它的返回值传递给了另一个函数。这种做法是错误的。当使用mysql\_query()函数的时候，如果它后面的函数必须在它调用成功的情况下才有意义，我们就必须对它的返回值进行检查。

即使mysql\_query()调用返回的是一个有效的结果集标识符而不是FALSE，也必须对它做出正确的解释才能保证脚本不会出现错误。比如说，人们对结果集标识符经常会做出两种错误的解释：1）认为这个返回值是一个数据行计数值；2）认为这个返回值包含着数据库查询命令所检索到的数据。这两种解释都是错误的，请看下一节里的示例说明。

#### 1. 处理没有返回结果集的查询

对于那些对数据行进行修改的查询命令，mysql\_query()函数的返回值并不是一个数据行计数值。它只表明那个查询是成功了还是失败了，再没有其他含义了。如果想知道那条查询命

令影响了多少个数据行,就必须发出一个mysql\_affected\_rows()调用。假设需要从member数据表里把第149号会员的记录删掉并想知道删除操作是否成功,下面这种做法是错误的:

```
$result_id = mysql_query ("DELETE FROM member WHERE member_id = 149");
if (!$result_id)
    print ("member 149 was not deleted\n");
else
    print ("member 149 was deleted\n");
```

这段代码的错误在于它把\$result\_id当做一个数据行计数值了——它错误地认为返回值0 (FALSE) 表示查询命令已经执行成功,只不过没有删除任何数据行而已。可这个0返回值的真正含义却是那条查询命令根本就没有执行!

下面是对mysql\_query()函数的返回值做出的另一种错误解释:

```
$result_id = mysql_query ("DELETE FROM member WHERE member_id = 149");
if (!$result_id)
    print ("query failed\n");
else
    print ("member 149 was deleted\n");
```

这段代码能正确地区分出查询命令是否执行成功,但它仍是错误的——它错误地认为如果查询命令执行成功,就肯定会删掉指定的数据行。为什么说这也是错误的呢?因为DELETE语句的执行成功与否并不依赖于它是否真的删除了一个数据行。如果member数据表里恰好有一个ID号等于149的记录项,MySQL当然会删掉这个数据行并使mysql\_query()函数返回TRUE。可是,如果ID号等于149的那条记录项不存在,mysql\_query()函数也将返回TRUE——因为这次查询是合法的。如果想知道这次成功的查询是否真的删掉了指定的数据行,就必须发出一个mysql\_affected\_rows()调用。下面这段代码对结果集标识符做出的解释才是正确的:

```
$result_id = mysql_query ("DELETE FROM member WHERE member_id = 149");
if (!$result_id)
    print ("query failed\n");
else if (mysql_affected_rows () < 1)
    print ("no record for member 149 was found\n");
else
    print ("member 149 was deleted\n");
```

## 2. 处理有返回结果集的查询

在对那些会返回一些数据行的数据库查询命令进行处理的时候,必须牢记这样一个原则:只有在数据库查询命令执行成功的前提下才能去取回其结果集里的内容。人们很容易忘记这个过程必须要有两个步骤,尤其是在数据库查询命令的结果集里只有一个值的时候。请看下面这个用来确定member数据表里有多少个数据行的查询:

```
$result_id = mysql_query ("SELECT COUNT(*) FROM member");
print ("The member table has $result_id records\n");
```

这段代码是不正确的。它的错误在于想当然地认为因为结果集里仅有一项数据值,所以它肯定就是mysql\_query()函数的返回值。事实却并非如此。虽说必须通过\$result\_id才能访问数据



库查询命令的结果集，但它本身并不是数据库查询命令的查询结果。即使结果集里只有一项数据，也必须在执行完查询命令之后再发出一个数据行取回调用才能获得它。下面的代码演示了一种正确的做法：先判断mysql\_query()调用是否成功，然后再通过一个mysql\_fetch\_row()调用把查询结果取回到变量\$row里；只有在这两个函数都调用成功的前提下，这段代码才会去打印COUNT(\*)的值：

```
$result_id = mysql_query ("SELECT COUNT(*) FROM member");
if (!$result_id || !($row = mysql_fetch_row ($result_id)))
    print ("query failed\n");
else
    print ("The member table has $row[0] records\n");
```

另一种正确的做法主要用在结果集里会有多个记录项的场合。在这类场合，通常要使用一个循环来取回各有关数据行。请看下面这段演示代码：

```
$result_id = mysql_query ("SELECT * FROM member");
if (!$result_id)
    print ("query failed\n");
else
{
    printf ("number of rows returned: %d\n", mysql_num_rows ($result_id));
    # fetch each row in result set
    while ($row = mysql_fetch_row ($result_id))
    {
        # print values in row, separated by commas
        for ($i = 0; $i < mysql_num_fields ($result_id); $i++)
        {
            if ($i > 0)
                print ("," );
            print ($row[$i]);
        }
        print ("\n");
    }
    mysql_free_result ($result_id);
}
```

在上面这段代码里，如果数据库查询命令执行失败，mysql\_query()调用的返回值就将是FALSE，我们将只打印一条相应的出错信息；如果数据库查询命令执行成功，mysql\_query()调用的返回值就将是一个合法的结果集标识符，这个标识符可用于许多方面（但不能用做数据行计数值！）。结果集标识符的用途主要有以下几种：

- 把它传递给mysql\_num\_rows()函数以确定结果集里的数据行个数。
- 把它传递给mysql\_num\_fields()函数以确定结果集里的数据列个数。
- 把它传递给某个数据行取回例程以取回结果集里的连续数据行。上面那个例子使用的是mysql\_fetch\_row()函数，但还有其他选择，稍后介绍。
- 把它传递给mysql\_free\_result()函数，让PHP释放这个结果集以及与之相关联的各种资源。

在mysql\_query()调用成功地执行了一个SELECT查询之后,我们就可以利用PHP提供的数据行取回函数(见表8-1)来处理它的结果集了。这些函数中的每一个都要使用一个给定的结果集标识符来作为自己的输入参数,当到达结果集里的最后一个数据行时,这些函数都将返回FALSE。

表8-1 PHP的数据行取回函数

函数名称	返回值
mysql_fetch_row()	一个数组,元素要通过数值下标来访问
mysql_fetch_assoc()	一个数组,元素要通过字符串值下标(即关联下标)来访问
mysql_fetch_array()	一个数组,元素可以通过数值下标或字符串值下标来访问
mysql_fetch_object()	一个对象,元素被当做该对象的属性来访问

最基本的调用是mysql\_fetch\_row(),它将返回结果集里的下一个数据行作为一个数组。这个数组里的元素要通过一个数值型下标来访问。下标的取值范围是0到mysql\_num\_fields()-1。下面的例子演示了mysql\_fetch\_row()函数的用法——利用一个简单的循环结构依次取回并打印出结果集里的每一个数据行,每一行里的各项数据用制表符间隔:

```
$query = "SELECT * FROM president";
$result_id = mysql_query ($query)
    or die ("Query failed");
while ($row = mysql_fetch_row ($result_id))
{
    for ($i = 0; $i < mysql_num_fields ($result_id); $i++)
    {
        if ($i > 0)
            print ("\t");
        print ($row[$i]);
    }
    print ("\n");
}
mysql_free_result ($result_id);
```

只要mysql\_fetch\_row()能取回一个数据行,变量\$row就将被赋值为一个数组。这个数组里的元素要用\$row[\$i]的形式来访问,其中的\$i是数值形式的数据列下标。如果想知道这个被表示为数组的数据行里有多少个元素,就应该把结果集标识符传递到mysql\_num\_fields()函数里去。看到这里,有些读者可能会问:把\$row传递到PHP专门用来统计一个数组里有多少个元素的count()函数里不就行了吗?为什么非得使用mysql\_num\_fields()函数呢?因为PHP会把包含在数据库查询结果集里的NULL值表示为unset值(即未定义值),而PHP 3中的count()函数在统计数组中的元素个数时却不会把unset值也计算在内。换句话说,用count()函数统计出来的结果集元素个数不一定准确。因此还是使用mysql\_num\_fields()函数,它是专门用于此的。

取回一个数组的另一种办法是把它赋值给一个变量列表。比如说,假设想把last\_name和first\_name数据列(即会员们的姓氏和名字)直接取到变量\$ln和\$fn里,然后再按名字在前、姓氏在后的顺序把会员们的姓名打印出来,就可以像下面这样做:

```

$query = "SELECT last_name, first_name FROM president";
$result_id = mysql_query ($query)
    or die ("Query failed");
while (list ($ln, $fn) = mysql_fetch_row ($result_id))
    printf ("%s %s\n", $fn, $ln);
mysql_free_result ($result_id);

```

变量可以是任何一个合法的名字，但它们在list()里的顺序必须与在数据库查询命令中选取的数据列顺序相对应。

表8-1里的第二个数据行取回函数mysql\_fetch\_assoc()也能以数组的形式返回结果集里的下一个数据行，但这个数组里的元素需要通过关联下标（即字符串值下标）来访问，各元素的名字（即下标）就是在数据库查询命令中给出的数据列的名字：

```

$query = "SELECT last_name, first_name FROM president";
$result_id = mysql_query ($query)
    or die ("Query failed");
while ($row = mysql_fetch_assoc ($result_id))
    printf ("%s %s\n", $row["first_name"], $row["last_name"]);
mysql_free_result ($result_id);

```

与其他几个数据行取回函数相比，mysql\_fetch\_assoc()函数是比较新的，它最早出现于PHP 4.0.3版本。

表8-1里的第三个数据行取回函数mysql\_fetch\_array()也能以数组的形式返回结果集里的下一个数据行，但这个数组里的元素既可以通过数值下标来访问，也可以通过字符串值下标来访问。换句话说，既可以通过元素在数组中的序号来访问它们，也可以通过它们的名字来访问它们：

```

$query = "SELECT last_name, first_name FROM president";
$result_id = mysql_query ($query)
    or die ("Query failed");
while ($row = mysql_fetch_array ($result_id))
{
    printf ("%s %s\n", $row[1], $row[0]);
    printf ("%s %s\n", $row["first_name"], $row["last_name"]);
}
mysql_free_result ($result_id);

```

mysql\_fetch\_array()函数所返回的信息是mysql\_fetch\_row()和mysql\_fetch\_assoc()两个函数所返回的信息混杂在一起而构成的一个组合。除此之外，这几个函数在性能方面的差异完全可以忽略不计，可以随意发出mysql\_fetch\_array()调用而不会对性能产生任何值得考虑的影响。

最后一个数据行取回函数，即mysql\_fetch\_object()，将把结果集里的下一个数据行返回为一个对象。也就是说，数据行里的元素必须通过\$row->col\_name语法来访问。比如说，如果想检索president数据表里的last\_name和first\_name值（即美国总统们的姓氏和名字），就可以用下面这样的代码来访问这两个数据列：

```

$query = "SELECT last_name, first_name FROM president";
$result_id = mysql_query ($query)
    or die ("Query failed");

```

```
while ($row = mysql_fetch_object ($result_id))
    printf ("%s %s\n", $row->first_name, $row->last_name);
mysql_free_result ($result_id);
```

可是, 如果查询命令里有一个计算出来的输出列该怎么办呢? 请看下面这个查询, 它将把一个表达式的计算结果返回为自己的查询结果:

```
SELECT CONCAT(first_name, ' ', last_name) FROM president
```

这类查询的结果集是不适合用mysql\_fetch\_object()函数来取回的: 所选取数据列的名字就是表达式本身, 可这并不是一个合法的属性名。不过, 可以通过给数据列安排一个别名 (alias) 的办法来提供一个合法的名字。请看下面这个查询, 它给输出列安排了一个别名叫full\_name:

```
SELECT CONCAT(first_name, ' ', last_name) AS full_name FROM president
```

现在, 如果仍打算用mysql\_fetch\_object()函数来取回这个查询命令的结果集的话, 就可以通过\$row->full\_name语法来访问那个输出列了。

#### 8.1.4 返回结果里NULL值的检测

PHP把数据库查询命令的结果集里的NULL值表示为unset值 (即未定义值)。我们可以用PHP函数isset()来检查某个SELECT查询所返回的数据列里是否包含有NULL值。请看下面这段用来打印member数据表里的电子邮件地址的代码, 当某个会员的电子邮件地址是NULL值时, 它将打印一条 “No email address available” (没有电子邮件地址) 信息:

```
$query = "SELECT last_name, first_name, email FROM member";
$result_id = mysql_query ($query)
    or die ("Query failed");
while (list ($last_name, $first_name, $email) = mysql_fetch_row ($result_id))
{
    printf ("Name: %s %s, Email: ", $first_name, $last_name);
    if (!isset ($email))
        print ("no email address available");
    else
        print ($email);
    print ("\n");
}
mysql_free_result ($result_id);
```

那么, 能不能用PHP语言中的empty()函数来检测NULL值呢? 不能, 因为empty()函数对NULL值和空字符串的返回值是一样的。换句话说, empty()函数不能把NULL值和空字符串区分开来, 所以不太适合用来检测NULL值。

如果使用的是PHP 4, 还可以用PHP语言中的NULL常数和 “===” 操作符 (注意: 是连续三个等号) 来检测NULL值, 如下所示:

```
$query = "SELECT last_name, first_name, email FROM member";
$result_id = mysql_query ($query)
    or die ("Query failed");
```

```

while (list ($last_name, $first_name, $email) = mysql_fetch_row ($result_id))
{
    printf ("Name: %s %s, Email: ", $first_name, $last_name);
    if ($email === NULL)
        print ("no email address available");
    else
        print ($email);
    print ("\n");
}
mysql_free_result ($result_id);

```

### 8.1.5 出错处理

PHP提供了三种可以用在基于MySQL的脚本里的出错处理机制，其中两种可以用来对多种出错情况进行处理，还有一种则是专门用来对与MySQL操作有关的出错情况进行处理的。第一种出错处理机制是用“@”操作符来抑制PHP函数可能产生的任何出错信息。其实我们已经见过“@”操作符的用法了，在前面调用mysql\_connect()函数的时候，就是用“@”操作符来防止来自该函数的出错信息出现在将被送往客户浏览器的页面里的：

```
$conn_id = @mysql_connect ("cobra.snake.net", "sampadm", "secret");
```

第二种出错处理机制是用PHP函数error\_reporting()来打开或者关闭指定级别（见表8-2）的出错报告功能。

表8-2 PHP的出错处理级别

出错处理级别	该级别所报告的出错信息类型
E_ERROR	一般性的函数调用出错信息
E_WARNING	一般性的警告信息
E_PARSE	来自PHP分析器的出错信息
E_NOTICE	通知性信息
E_CORE_ERROR	来自核心引擎的出错信息
E_CORE_WARNING	来自核心引擎的警告信息
E_COMPILE_ERROR	来自编译器的出错信息
E_COMPILE_WARNING	来自编译器的警告信息
E_USER_ERROR	用户生成的出错信息
E_USER_WARNING	用户生成的警告信息
E_USER_NOTICE	用户生成的通知性信息
E_ALL	所有错误

用error\_reporting()函数来控制出错报告功能的具体做法是：以想激活的各出错处理级别的二进制或操作（OR）的运算结果为输入参数去调用这个函数。一般来说，关闭E\_ERROR和E\_WARNING两个出错处理级别就已经足以抑制来自MySQL函数的出错信息了，如下所示（这里的做法是只激活E\_PARSE和E\_NOTICE两个出错处理级别）：

```
error_reporting (E_PARSE | E_NOTICE);
```



对应于PHP分析器出错的E\_PARSE级警告信息通常用不着关闭,如果关闭了它,对脚本所做的改动就不容易调试了。E\_NOTICE级警告信息大都可以忽略,但它们往往预示着脚本存在着一些值得注意的小毛病,所以还是让它处于激活状态比较好。一般用不着关心其他的出错处理级别,如有必要,可以激活E\_ALL级别来查看所有的出错信息。

需要提醒大家注意的是,PHP 3只支持4种出错处理级别,而且这些级别必须通过它们各自的编号来引用,不能使用符号常数来引用它们(见表8-3)。

表8-3 PHP 3的出错处理级别

出错处理级别	该级别所报告的出错信息类型
1	一般性的函数调用出错信息
2	一般性的警告信息
4	来自PHP分析器的出错信息
8	通知性信息

第三种出错处理机制是使用mysql\_error()和mysql\_errno()函数,这两个函数能够把由MySQL服务器返回的出错信息报告出来。它们与C API里的同名函数调用差不多:mysql\_error()函数返回的是一个出错信息字符串(空字符串表示没有出错),mysql\_errno()函数返回的是一个出错代码(0表示没有出错)。这两个函数都要以一个连接标识符作为其输入参数(可以在用来访问MySQL数据库的PHP脚本里同时打开多条与MySQL服务器的连接,但是这些连接必须通过不同的连接标识符加以区分),它们返回的是指定连接上最后一个被调用的、有状态信息可供返回的那个函数(有些PHP函数不返回状态信息)的出错信息。这两个函数的连接标识符输入参数都是可选的,如果没有给出,就将使用最后一个被打开的MySQL连接。比如说,可以像下面这样报告来自mysql\_query()调用的出错信息:

```
$result_id = mysql_query ($query);
if (! $result_id)
{
    print ("errno: " . mysql_errno() . "\n");
    print ("error: " . mysql_error() . "\n");
}
```

在C API里,即使连接MySQL服务器的尝试没有成功,也可以利用mysql\_error()和mysql\_errno()函数来获得出错信息。在PHP脚本里也是如此,但只有PHP 4.0.6及更高的版本才支持这种做法。在PHP 4.0.6之前的版本里,如果连接MySQL服务器的尝试没有成功,该连接上的mysql\_error()和mysql\_errno()调用就不会返回有用的出错信息。(换句话说,如果连接MySQL服务器的尝试没有成功,就无法通过mysql\_error()和mysql\_errno()调用来了解问题的原因。)在这类场合,如果想报告连接失败的具体原因而不是一句简单的“连接失败”之类的信息,就必须采取一些特殊的措施,关于这方面的具体做法请参见附录H。

为简单起见,这一章里的脚本在执行出错时都只打印一条比较简单的信息,比如“Query failed”(查询失败)等等。但是,当编写脚本的时候,在代码里增加一些mysql\_error()调用往往有助于找到问题的具体原因。

### 8.1.6 引号问题

当在PHP脚本里构造数据库查询命令字符串的时候，一定要给引号问题以足够的重视，就像我们在使用C和Perl等其他程序设计语言来开发MySQL脚本时一样。虽然各种程序设计语言里的函数名称会有所不同，但它们在解决引号问题方面的基本思路却是一致的。我们先来看一个例子。假设构造一条用来把一个新记录插入到数据表里去的查询命令。在构造查询命令字符串的时候，给那些将被插入到字符串数据列里的数据值都加上了引号，如下所示：

```
$last = "O'Malley";
$first = "Brian";
$expiration = "2002-9-1";
$query = "INSERT INTO member (last_name,first_name,expiration)"
        . " VALUES('$last','$first','$expiration')";
```

这里的问题是有一项数据（O'Malley）本身就带有一个单引号，如果把这个查询命令发送给MySQL服务器，就会导致语法错误。为了解决这一问题，在C语言程序里，可以调用mysql\_real\_escape\_string()或mysql\_escape\_string()函数；在Perl DBI脚本里，可以调用quote()函数。在PHP脚本里，可以通过一个名为addslashes()的函数来实现类似的目标。比如说，函数调用addslashes("O'Malley")将返回字符串值“O\Malley”。上例中的引号问题可以用下面这段代码来解决：

```
$last = addslashes ("O'Malley");
$first = addslashes ("Brian");
$expiration = addslashes ("2002-9-1");
$query = "INSERT INTO member (last_name,first_name,expiration)"
        . " VALUES('$last','$first','$expiration')";
```

DBI模块里的quote()函数会在字符串的两端加上一对引号。但PHP里的addslashes()函数却没有这样做，还得由你亲自在数据库查询命令字符串里给各有关数据项加上一对引号才行。这个办法适用于除unset值以外的各种数据，因为需要把PHP语言里的unset值在MySQL查询命令字符串里转换为一个不带任何引号的单词——NULL。这个问题稍后介绍。

作为addslashes()函数的替代物，PHP从4.0.3版本开始新增了一个名为mysql\_escape\_string()的函数，从4.3.0版本开始又新增了一个名为mysql\_real\_escape\_string()的函数：

```
$escaped_str = mysql_escape_string ($str);
$escaped_str = mysql_real_escape_string ($str, $conn_id);
```

这两个新增函数分别以MySQL C API里的同名函数为基础。因为mysql\_real\_escape\_string()函数要求使用指定MySQL服务器连接的字符集对数据进行编码，所以它多出了一个连接标识符输入参数，如果省略这个参数，这个函数将使用当前连接的字符集。

为了让PHP脚本在各种PHP版本下都能运行，就必须根据具体的PHP版本选用正确的字符串转义函数来处理数据库查询命令中的字符串数据值。以人工方式来进行这种“三选一”的工作未免过于麻烦，所以决定利用现在这个机会编写一个名为quote\_value()的辅助函数并把它放到sampdb.php库文件里去。这个函数能够根据PHP的具体版本选出一个最合适的编码函数来对字

符串数据值进行转义处理,它使用了function\_exists()函数来检查某个字符串转义函数是否存在。下面就是quote\_value()函数的代码:

```
function quote_value ($str)
{
    if (!isset ($str))
        return ("NULL");
    if (function_exists ("mysql_real_escape_string"))
        return ("'" . mysql_real_escape_string ($str) . "'");
    if (function_exists ("mysql_escape_string"))
        return ("'" . mysql_escape_string ($str) . "'");
    return ("'" . addslashes ($str) . "'");
}
```

除能挑选出最合适的编码函数外,quote\_value()函数还解决了前面提到的需要把PHP语言中的unset值转换为MySQL数据库里的NULL值的问题。如果它的输入参数是一个unset值,quote\_value()将返回一个不带任何引号的单词NULL;否则,它将返回一个经适当转义处理并在首尾两端加上了单引号的结果字符串。也就是说,如果利用quote\_value()函数来构造数据库查询命令的话,只要把它的返回值直接放到查询命令字符串里的适当位置就行了,用不着再亲自去添加额外的引号,如下所示:

```
$last = quote_value ("O'Malley");
$first = quote_value ("Brian");
$expiration = quote_value ("2002-9-1");
$query = "INSERT INTO member (last_name,first_name,expiration)"
        . " VALUES($last,$first,$expiration)";
```

以上讨论只能解决我们在构造将被送往MySQL服务器去的查询命令字符串时可能遇到的各种引号问题,但在生成将被显示为Web页面的输出内容时也同样会遇到一些引号问题。只要准备生成为HTML文本或URL地址的某个字符串里可能包含有相应的特殊字符(注意,HTML文本和URL地址里的特殊字符并不完全相同),就应该对它们进行适当的编码。这两项编码工作可以用PHP提供的htmlspecialchars()和urlencode()函数来分别完成,它们与第7章介绍的CGI.pm模块的escapeHTML()和escape()方法很相似。

## 8.2 PHP脚本实战

本章的后续内容将解决第1章里提出的一些目前仍未完成的目标:

- 考试记分项目:编写一个用来录入和编辑考试分数的脚本。
- 美国历史研究会项目:在研究会的Web站点上提供一个关于美国总统生平事迹的小测验。我们希望这个小测验是交互式的,测验题采用即时的方法生成。
- 美国历史研究会项目:使研究会的会员能够通过网络以在线方式来修改他们的会员名录资料。这一方面能使有关信息保持最新,同时也减少了研究会秘书(也就是你)的工作量。

这些脚本都需要生成多个彼此关联的Web页面,所以这些脚本在它们各自的前、后两次调用中需要通过嵌在Web页面里的信息进行通信。如果你不熟悉Web页面之间的通信机制,请参阅

第7.4.2节的有关内容。

### 8.2.1 考试记分项目：考试分数的录入

在这一小节，我们将把注意力集中到考试记分项目并编写一个score\_entry.php脚本来管理学生们的考试分数。把与该项目有关的Web页面全都放在Apache文档树里的gp目录里，这个目录在站点上对应于下面这个URL地址：

`http://www.snake.net/gp/`

这个目录现在还空无一物，请求这个URL地址的访问者目前只能看到一条“Page not found”（网页未找到）信息或者一个空白的目录清单页面，所以我们现在的当务之急是要在gp目录里创建一个简短的index.php脚本来充当考试记分项目的主页。下面这个脚本足以应付眼前的需要。这个脚本生成的Web页面里有两个超链接，一个指向第7章里为考试记分项目编写的score\_browse.pl脚本，另一个指向即将编写的score\_entry.php脚本：

```
<?php
# Grade-Keeping Project home page

include "sampdb.php";

$title = "Grade-Keeping Project";
html_begin ($title, $title);
?>

<p>
<a href="/cgi-bin/score_browse.pl">View</a> test and quiz scores
</p>
<p>
<a href="score_entry.pl">Enter or edit</a> test and quiz scores
</p>

<?php
html_end ();
?>
```

我们将要设计和实现的score\_entry.php脚本具备两大功能，一是使我们能够录入一组考试分数，二是使我们能够对现有的各组考试分数进行编辑。它提供的录入功能将使我们得以把学生们的考试分数添加到数据库里去，而它提供的编辑功能使我们能够在今后对考试分数进行修改，比如把因生病或其他原因而缺考的学生们的补考成绩录入数据库或者改正我们输错的考试成绩等等。下面是这个脚本的概念性框架：

- 在初始页面上，这个脚本将显示一份已知考试事件的清单，既可以点选一个现有事件，也可以去创建一个新的考试事件。
- 如果选择的是创建一个新的考试事件，脚本将在下一个页面里要求你给出一个考试日期和考试事件的类型（考试或测验）。在把新事件添加到数据库里之后，脚本将重新显示考试



事件清单页面,新添加的考试事件也将出现在这个页面里。

- 如果从清单里选择了一个现有的考试事件,脚本将显示一个考试分数录入页面并在页面上给出该次考试事件的ID编号、考试日期、一份学生名单和一个Submit(提交)按钮。学生名单的每一行列出了一位学生的姓名和他这次考试的分数。如果选择的是一个新的考试事件,学生们的考试分数将都是空白。如果选择的是一个现有的考试事件,考试分数将是此前录入的那些数字。当录入或者修改完学生们的考试分数后,点击Submit按钮,脚本就会把考试分数输入到score数据表或者完成对现有分数的修改。

在开始实现score\_entry.php脚本之前,先讨论一下输入参数在PHP脚本里的工作原理。这个脚本需要完成几个不同的动作,这意味着它必须在页面之间传递一个状态值,通过这个状态值去告诉脚本在每一次的执行过程中应该做什么事情。传递这个状态值的办法之一是在URL地址的尾部追加一个参数。比如说,可以像下面这样在这个脚本的URL地址尾部加上一个名为action的参数:

```
http://www.snake.net/gp/score_entry.php?action=value
```

参数值也可以来自用户提交的某个表单的内容。作为表单提交动作的组成部分,由用户的浏览器返回的表单里的每一个字段都有一个名字和一个值。

PHP把来自Web的输入放在几个特殊的数组里供脚本使用。被编码在URL地址尾部以及通过GET方法发送回来的参数将放在一个名为\$HTTP\_GET\_VARS的全局数组里,通过POST方法接收到的参数(例如,具有POST的method属性值的表单的内容)将放在一个名为\$HTTP\_POST\_VARS的全局数组里。这两个数组都是关联数组,元素的键值就是参数的名字。比如说,如果action参数是以被编码在URL地址尾部的方式发送回来的,就可以在PHP脚本里通过变量\$HTTP\_GET\_VARS["action"]来访问和使用它。如果某个表单里有一个名为address的字段并且这个表单是以POST请求的方式被提交的,就可以在PHP脚本里通过变量\$HTTP\_POST\_VARS["address"]来访问和使用它。

即使表单有多个字段,PHP脚本也可以访问到各字段的参数值。比如说,假设某表单里包含着名为name和address的两个字段,当用户提交这个表单的时候,他的Web服务器将调用一个脚本来处理这个表单的内容。那么,如果这个表单是通过GET请求提交给Web服务器的,脚本只需检查变量\$HTTP\_GET\_VARS["action"]和\$HTTP\_GET\_VARS["action"]的取值就能知道用户在表单里输入了什么样的值;而如果这个表单是通过POST请求提交给Web服务器的,脚本只需检查变量\$HTTP\_POST\_VARS["action"]和\$HTTP\_POST\_VARS["action"]的取值就能知道用户在表单里输入了什么样的值。不过,如果表单里有多个字段,让每个字段都有一个独一无二的名字就不太容易做到。幸好PHP能够方便地把表单里的参数传递为一系列数组。比如说,如果使用的字段名是x[0]、x[1]等等,PHP就将把它们保存在\$HTTP\_GET\_VARS["x"]或\$HTTP\_POST\_VARS["x"]数组里。如果把这个数组赋值给变量\$x,就可以通过\$x[0]、\$x[1]等构造来访问这个数组的各个元素了。

在大多数场合,并不需要关心那些参数是以GET方式还是以POST方式提交给Web服务器的,所以现在编写一个辅助函数script\_param()来完成参数值的提取工作,它将根据我们给定的一个参数名去检查那两个全局数组并把该参数的值从来自Web的输入里提取出来;若来自Web的输入



里没有这个参数，则返回一个unset值。下面是这个函数的代码：

```
function script_param ($name)
{
    global $HTTP_GET_VARS, $HTTP_POST_VARS;

    unset ($val);
    if (isset ($HTTP_GET_VARS[$name]))
        $val = $HTTP_GET_VARS[$name];
    else if (isset ($HTTP_POST_VARS[$name]))
        $val = $HTTP_POST_VARS[$name];
    # return @$val rather than $val to prevent "undefined value"
    # messages in case $val is unset and warnings are enabled
    return (@$val);
}
```

注意，script\_param()函数用关键字global把那两个数组明确地声明为全局变量。这是因为，在PHP脚本里，如果想在某个子例程的作用范围内使用PHP全局变量，就必须在这个子例程里用global关键字对它做出声明；否则，PHP就会优先使用属于这个子例程的、与那个全局变量同名的局部变量。也就是说，如果在全局范围内（比如脚本的主体部分里），可以不加声明地使用PHP全局变量；但如果在非全局范围内（比如脚本的某个子函数里），global关键字将向PHP表明你想访问的是一个全局变量而不是该函数作用范围内的一个恰好与PHP全局变量同名的局部变量。script\_param()函数还在return()语句的前面加上了一个“@”操作符以抑制可能产生的出错信息。（如果来自Web的输入里没有某个参数，script\_param()将返回一个unset值。如果脚本恰好激活了相应的出错处理级别的话，返回一个unset值就会导致script\_param()产生一条警告信息，而这种情况是我们不希望看到的。）

PHP 4.1引入了两个新的参数数组：\$\_GET和\$\_POST。这两个数组与前面介绍的两个全局参数数组\$HTTP\_GET\_VARS和\$HTTP\_POST\_VARS很相似，但前者是两个超全局数组。这句话的意思是：可以在任何作用范围内访问这两个新数组而无需事先使用global关键字对它们做出声明。下面对script\_param()函数稍微做些修改，让它尽可能地使用这两个比较新的超全局数组（如果它们可用的话）从来自Web的输入里提取参数值：

```
function script_param ($name)
{
    global $HTTP_GET_VARS, $HTTP_POST_VARS;

    unset ($val);
    if (isset ($_GET[$name]))
        $val = $_GET[$name];
    else if (isset ($_POST[$name]))
        $val = $_POST[$name];
    else if (isset ($HTTP_GET_VARS[$name]))
        $val = $HTTP_GET_VARS[$name];
    else if (isset ($HTTP_POST_VARS[$name]))
        $val = $HTTP_POST_VARS[$name];
}
```

```

    if (isset ($val) && get_magic_quotes_gpc ())
        $val = remove_backslashes ($val);
    # return @$val rather than $val to prevent "undefined value"
    # messages in case $val is unset and warnings are enabled
    return (@$val);
}

```

在sampdb发行版本中的sampdb.php库文件里看到的script\_param()函数就是上面这个改进版本。有了它,不管来自Web的输入参数存放在哪一个数组里,脚本都可以简单地通过这些参数的名字直接去访问它们的值。在这个改进版的script\_param()函数里,除增加了对\$\_GET和\$\_PUT数组是否存在的检查外,还多了一项改动:在提取出一个参数值以后,还要把这个参数值传递到remove\_backslashes()函数里去。增加这项改动的目的是为了使这个函数能够适应PHP初始化文件用下面这条语句把配置选项magic\_quotes\_gpc设置为激活状态的配置情况:

```
magic_quotes_gpc = On;
```

如果这个选项被激活,PHP就会在从Web返回的输入参数值里自动添加一些反斜线字符(\)以对其中的特殊字符(比如引号或反斜线字符等)进行转义。这些额外的反斜线字符会增加我们检查参数值是否有效这一工作的难度,所以用remove\_backslashes()函数把它们去除掉。下面是remove\_backslashes()函数的实现代码。在PHP 4里,来自Web的输入参数有可能返回为一系列多层嵌套的数组,所以这里使用了一个递归算法。

```

function remove_backslashes ($val)
{
    if (!is_array ($val))
        $val = stripslashes ($val);
    else
    {
        reset ($val);
        while (list ($k, $v) = each ($val))
            $val[$k] = remove_backslashes ($v);
    }
    return ($val);
}

```

### 来自Web的输入参数与PHP配置选项register\_globals

读者对PHP的配置选项register\_globals可能不会感到陌生,它的作用是使来自Web的输入参数直接被注册为脚本里的变量。比如说,一个名为x的表单字段或URL参数将直接保存到脚本中一个名为\$x的变量里。这一做法虽然方便,但同时也意味着Web客户有能力直接在你的脚本里创建出一些不希望的变量来。这是一种安全漏洞,所以PHP的开发者们现在建议最好禁用register\_globals选项。在编写script\_param()函数的时候,本书特意只使用了PHP专为脚本提供输入参数而准备的数组,这既增加了安全性,也使它能够工作在register\_globals选项的任何一种设置情况下。

现在,既然我们已经有了提取Web输入参数这一辅助工具,下面就使用这个工具来编写score\_entry.php脚本。这个脚本需要在它自己的前后两次调用中交流一些信息。我们将使用一个名为action的参数来做这件事情,这个参数的值可以通过下面的PHP代码来获得:

```
$action = script_param ("action");
```

如果来自Web的输入里没有给出这个参数,就说明这是脚本的首次调用;否则,脚本将根据变量\$action的取值来决定将要做哪些事情。下面是score\_entry.php脚本的基本框架:

```
<?php
# score_entry.php - Score Entry script for grade-keeping project

include "sampdb.php";

# define action constants
define ("SHOW_INITIAL_PAGE", 0);
define ("SOLICIT_EVENT", 1);
define ("ADD_EVENT", 2);
define ("DISPLAY_SCORES", 3);
define ("ENTER_SCORES", 4);

# ... put input-handling functions here ...

$title = "Grade-Keeping Project -- Score Entry";
html_begin ($title, $title);

sampdb_connect()
    or die ("Cannot connect to database server");

# determine what action to perform (the default if
# none is specified is to present the initial page)

$action = script_param ("action");
if (!isset ($action))
    $action = SHOW_INITIAL_PAGE;

switch ($action)
{
case SHOW_INITIAL_PAGE:      # present initial page
    display_events ();
    break;
case SOLICIT_EVENT:          # ask for new event information
    solicit_event_info ();
    break;
case ADD_EVENT:              # add new event to database
    add_new_event ();
    display_events ();
    break;
```

```

case DISPLAY_SCORES:          # display scores for selected event
    display_scores ();
    break;
case ENTER_SCORES:            # enter new or edited scores
    enter_scores ();
    display_events ();
    break;
default:
    die ("Unknown action code ($action)");
}

html_end ();
?>

```

变量\$action有几种可能的取值,而我们用了一个switch语句来对它进行检测。PHP语言里的switch语句与C语言里的switch很相似,它在这里的作用是判断脚本应该采取什么动作并调用有关的函数来完成这个动作。为了增加代码的可读性,在switch语句里使用了一些符号化的动作名称,这些符号是在脚本的开头部分利用PHP语言的define()构造定义的。

下面,我们将依次分析并实现这个脚本里的各个动作。第一个动作,即display\_events()函数,将显示一个考试事件清单,是通过发出MySQL查询命令去检索event数据表的数据行的办法来获得和显示有关信息的。这份清单里的每一行列出了相应的考试事件ID编号、考试日期和考试事件类型(考试或测验)。Web页面里的事件ID都是一些超链接,点选它们就能对学生们在给定考试事件中的分数进行编辑。此外,在这个初始页面的末尾,display\_events()函数还在考试清单的下方提供了一个用来创建新考试事件的超链接。下面是这个函数的代码:

```

function display_events ()
{
    print ("Select an event by clicking on its number, or select\n");
    print ("New Event to create a new grade event:<br /><br />\n");
    $query = "SELECT event_id, date, type FROM event ORDER BY event_id";
    $result_id = mysql_query ($query)
        or die ("Cannot execute query");
    print ("<table border='1'\n");

    # Print a row of table column headers

    print ("<tr>\n");
    display_cell ("th", "Event ID");
    display_cell ("th", "Date");
    display_cell ("th", "Type");
    print ("</tr>\n");

    # Present list of existing events. Associate each event id with a
    # link that will show the scores for the event; use mysql_fetch_array()
    # to fetch each row so that its columns can be referred to by name.

    while ($row = mysql_fetch_array ($result_id))
    {

```

```

print("<tr>\n");
$url = sprintf ("%s?action=%s&event_id=%s",
                script_name (),
                urlencode (DISPLAY_SCORES),
                urlencode ($row["event_id"]));
display_cell ("td",
              "<a href=\""$url\"">"
                . htmlspecialchars ($row["event_id"])
                . "</a>",
              FALSE);
display_cell ("td", $row["date"]);
display_cell ("td", $row["type"]);
print("</tr>\n");
}

# Add one more link for creating a new event

print("<tr align=\"center\">\n");
$url = sprintf ("%s?action=%s",
                script_name (),
                urlencode (SOLICIT_EVENT));
display_cell ("td colspan=\"3\"",
              "<a href=\""$url\"">" . "Create New Event" . "</a>",
              FALSE);
print("</tr>\n");

print("</table>\n");
}

```

这个页面里超链接将导致score\_entry.php脚本被再次调用。我们在构造这些超链接的URL地址时使用了一个名为script\_name()的函数(可以在sampdb发行版本的sampdb.php文件里找到这个函数),它的用途是确定脚本自身的路径名。有了script\_name()函数,我们就不必把脚本的文件名硬编码在代码里,如果把脚本的文件名硬编码在代码里而以后又重新命名了这个文件,这个脚本就无法正常工作了。

类似于前面编写的script\_param()函数,script\_name()函数也需要访问几个PHP全局数组。但这两个函数所使用的数组是不同的,因为脚本的文件名是Web服务器所提供的信息的一部分。在来自Web的输入参数里是找不到这个信息的。下面是script\_name()函数的代码:

```

function script_name ()
{
    global $HTTP_SERVER_VARS, $PHP_SELF;

    if (isset ($_SERVER["PHP_SELF"]))
        return ($_SERVER["PHP_SELF"]);
    if (isset ($HTTP_SERVER_VARS["PHP_SELF"]))
        return ($HTTP_SERVER_VARS["PHP_SELF"]);
    return ($PHP_SELF);
}

```



display\_events()函数又调用了子函数display\_cell()来生成event数据表里的各项数据:

```
# Display a cell of an HTML table. $tag is the tag name ("th" or "td"
# for a header or data cell), $value is the value to display, and
# $encode should be true or false, indicating whether or not to perform
# HTML-encoding of the value before displaying it. $encode is optional,
# and is true by default.

function display_cell ($tag, $value, $encode = TRUE)
{
    if ($value == "") # is the value empty or unset?
        $value = "&nbsp;";
    else if ($encode) # perform HTML-encoding if requested
        $value = htmlspecialchars ($value);
    print("<$tag>$value</$tag>\n");
}
```

如果在display\_events()函数生成并显示的Web表格里点选了“Create New Event”(创建新考试事件)链接, score\_entry.php脚本就将被再次调用并完成SOLICIT\_EVENT动作。这个动作将由solicit\_event\_info()函数负责完成, 这个函数会显示一个表单供你输入新考试事件的日期和类型(考试或测验)。下面是solicit\_event\_info()函数的代码:

```
function solicit_event_info ()
{
    printf("<form method=\"POST\" action=\"%s?action=%s\">\n",
        script_name (),
        urlencode (ADD_EVENT));
    print("Enter information for new grade event:<br /><br />\n");
    print("Date: ");
    print("<input type=\"text\" name=\"date\" value=\"\" size=\"10\" />\n");
    print("<br />\n");
    print("Type: ");
    print("<input type=\"radio\" name=\"type\" value=\"T\"");
    print(" checked=\"checked\" />Test\n");
    print("<input type=\"radio\" name=\"type\" value=\"Q\" />Quiz\n");
    print("<br /><br />\n");
    print("<input type=\"submit\" name=\"button\" value=\"Submit\" />\n");
    print("</form>\n");
}
```

在solicit\_event\_info()函数生成的表单里有一个用来输入考试日期的输入框、两个用来选择新事件是一次考试还是一次测验的单选按钮以及一个用来提交这个表单的Submit(提交)按钮。默认的事件类型是'T', 即这是一次考试。当填写好这份表单并提交它时, score\_entry.php脚本将被再次调用并完成ADD\_EVENT动作。这个动作将由add\_new\_event()函数负责完成, 这个函数将在event数据表里插入一个新的数据行, 这也是这个脚本与MySQL数据库所打的第一次交道。下面是add\_new\_event()函数的代码:

```

function add_new_event ()
{
    $date = script_param ("date"); # get date and event type
    $type = script_param ("type"); # entered by user

    if (empty ($date)) # make sure a date was entered, and in ISO format
        die ("No date specified");
    if (!preg_match ('/^\\d+\\D\\d+\\D\\d+$/ ', $date))
        die ("Please enter the date in ISO format (CCYY-MM-DD)");
    if ($type != "T" && $type != "Q")
        die ("Bad event type");

    $date = quote_value ($date);
    $type = quote_value ($type);
    if (!mysql_query ("INSERT INTO event (date,type) VALUES($date,$type)"))
        die ("Could not add event to database");
}

```

add\_new\_event()先调用script\_param()库例程来提取新事件创建表单里的date和type字段里对应的参数值。然后,为安全起见,进行了以下几项小检查:

- 来自date字段的日期参数值不允许为空,并且必须按ISO格式输入。对日期值是否符合ISO格式的检查是由preg\_match()函数通过模式匹配操作来实现的,如下所示:

```
preg_match ('/^\\d+\\D\\d+\\D\\d+$/ ', $date)
```

这里使用了单引号以防止美元字符(\$)和反斜线字符(\)被解释为特殊字符。如果输入的日期值由三组以非数字字符分隔的数字序列构成,这个测试的结果就将为真。虽说它无法做到万无一失,但这个测试很容易添加到脚本里,并且它也确实能捕获不少的常见错误。

- 来自type字段的事件类型参数值必须是event数据表中的type数据列允许使用的两个取值(即'T'和'Q')之一。

如果参数值满足以上检查条件,add\_new\_event()函数就会把一条新记录插入到event数据表里去。在构造查询命令字符串的时候,我们使用了quote\_value()库例程来完成引号的添加工作以确保插入到查询命令字符串里的各有关数据不会导致引号问题。注意,在构造并执行完数据库查询语句之后,add\_new\_event()函数先返回到了脚本的主体(它所在的switch...case子句),我们是在又调用了一次display\_events()函数后才退出这个switch...case子句的。这是为了让你能够在填写并提交了新事件创建表单之后立刻看到一份新的考试事件清单并开始录入学生们在新考试事件中的考试分数。

当在display\_events()函数生成的考试事件清单页面里选择了某个考试事件ID超链接时,score\_entry.php脚本将被再次调用并完成DISPLAY\_SCORES动作,这个动作将由display\_scores()函数负责完成。关联在这些超链接上的考试事件ID将被编码为一个event\_id参数从Web浏览器经Web服务器传递到这个脚本。display\_scores()函数将提取出这个参数的值并在检查确认它是一个整数之后,用它构造一个数据库查询命令去检索学生们在这次考试事件中取得

的分数。下面是display\_scores()函数的代码:

```
function display_scores ()
{
    # Get event ID number, which must look like an integer
    $event_id = script_param ("event_id");
    if (!preg_match ('/^\\d+$/', $event_id))
        die ("Bad event ID");

    # select scores for the given event
    $query = sprintf ("
        SELECT
            student.student_id, student.name, event.date,
            score.score AS score, event.type
        FROM student, event
        LEFT JOIN score ON student.student_id = score.student_id
            AND event.event_id = score.event_id
        WHERE event.event_id = %s
        ORDER BY student.name
    ", quote_value ($event_id));
    $result_id = mysql_query ($query)
        or die ("Cannot execute query");
    if (mysql_num_rows ($result_id) < 1)
        die ("No information was found for the selected event");

    printf ("<form method=\\\"POST\\\" action=\\\"%s?action=%s&event_id=%s\\\">\\n",
        script_name (),
        urlencode (ENTER_SCORES),
        urlencode ($event_id));

    # print scores as an HTML table

    $row_num = 0;
    while ($row = mysql_fetch_array ($result_id))
    {
        # print event info and table heading preceding the first row
        if ($row_num == 0)
        {
            printf ("Event ID: %s, Event date: %s, Event type: %s\\n",
                htmlentities ($event_id),
                htmlentities ($row["date"]),
                htmlentities ($row["type"]));
            print ("<br /><br />\\n");
            print ("<table border=\\\"1\\\">\\n");
            print ("<tr>\\n");
            display_cell ("th", "Name");
            display_cell ("th", "Score");
            print "</tr>\\n";
        }
    }
}
```

```

    }
    ++$row_num;
    print("<tr>\n");
    display_cell("td", $row["name"]);
    $col_val = sprintf("<input type=\"text\" name=\"score[%s]\",
                        htmlspecialchars($row[\"student_id\"]));
    $col_val .= sprintf(" value=\"%s\" size=\"5\" /><br />\n",
                        htmlspecialchars($row[\"score\"]));
    display_cell("td", $col_val, FALSE);
    print("</tr>\n");
}

print("</table>\n");
print("<br />\n");
print("<input type=\"submit\" name=\"button\" value=\"Submit\" />\n");
print("</form>\n");
}

```

为了把学生们在所选定的考试事件中取得的考试分数正确地检索出来，我们在display\_scores()函数里构造了一条将对多个数据表进行关联查询的数据库查询命令。请注意，这个查询并不是一个简单的数据表关联查询操作，因为那样会漏掉那些没有参加这次考试因而也就没有这次考试分数的学生；再进一步讲，如果所选取的是一个新考试事件，这种简单的关联查询操作将找不到任何记录，在Web浏览器里看到的将是一个空无一物的成绩表。因此，要想把每位学生的记录都检索出来而不管score数据表里是否有他的考试分数，必须使用一个LEFT JOIN操作。如果某个学生没有参加这次考试，他这次的考试分数在LEFT JOIN查询结果里就将是NULL。（第3.5.3节对我们在display\_scores()函数里构造和使用的这种LEFT JOIN查询进行了分析和说明。）

在检索出学生们的考试分数之后，display\_scores()函数将把它们全都显示在一个HTML表单里：学生们的考试分数分别对应着HTML表单中的一个输入字段，这些输入字段的名称是score[n]，而这个n是各位学生们的学号（即一个student\_id值）。当在Web页面上完成了考试分数的录入或编辑工作之后，只要提交这个表单，其中的考试分数就会送往数据库保存起来。当浏览器把这个表单发送回Web服务器时，其中所有的考试分数输入字段将作为一个名为score的参数被传递到score\_entry.php脚本里。这个score参数的值是一个数组，它的每一个元素分别对应着HTML表单中的一个考试分数输入字段。在脚本里，可以通过下面这个PHP表达式把score参数的值提取到一个变量里去：

```
$score = script_param ("score");
```

根据我们在display\_scores()函数里做出的安排，数组\$score各个元素的键值将是某一位学生的学号。这样，我们就能把通过HTML表单提交回来的考试分数与每一位学生对应起来了。表单的内容将由enter\_scores()函数负责处理，下面是这个函数的代码：

```

function enter_scores ()
{

```

```

# Get event ID number and array of scores for the event

$event_id = script_param ("event_id");
$score = script_param ("score");

if (!preg_match ('/^\\d+$/', $event_id)) # must look like integer
    die ("Bad event ID");

$invalid_count = 0;
$blank_count = 0;
$nonblank_count = 0;
reset ($score);
while (list ($student_id, $newscore) = each ($score))
{
    $newscore = trim ($newscore);
    if (empty ($newscore))
    {
        # if no score is provided for student in the form, delete any
        # score the student may have had in the database previously
        ++$blank_count;
        $query = sprintf ("
                                DELETE FROM score
                                WHERE event_id = %s AND student_id = %s
                                ",
                            quote_value ($event_id),
                            quote_value ($student_id));
    }
    else if (!preg_match ('/^\\d+$/', $newscore)) # must look like integer
    {
        ++$nonblank_count;
        $query = sprintf ("
                                REPLACE INTO score (event_id,student_id,score)
                                VALUES(%s,%s,%s)
                                ",
                            quote_value ($event_id),
                            quote_value ($student_id),
                            quote_value ($newscore));
    }
    else
    {
        ++$invalid_count;
        continue;
    }
    if (!mysql_query ($query))
        die ("score entry failed, event_id $event_id, "
            . "student_id $student_id");
}

```



```

printf ("Number of scores entered: %d<br />\n", $nonblank_count);
printf ("Number of scores missing: %d<br />\n", $blank_count);
printf ("Number of invalid scores: %d<br />\n", $invalid_count);
print ("<br />\n");
}

```

在enter\_scores()函数的循环语句里,我们先通过PHP函数each()从\$score数组中提取出每位学生的学号和与之关联的考试分数,然后对考试分数进行以下处理:

- 给定一个考试分数,如果它在去掉尾部多余的空白字符后是一个空值,enter\_scores()函数将简单地发出一条DELETE语句删除这位学生的这次考试分数记录。这么做的原因有两点:
  - 1) 如果score数据表里有这位学生这次考试的分数,就说明此前曾错误地给这位没有参加过考试的学生录入了一个考试分数,现在要把它改正过来;
  - 2) 如果score数据表里本来就没有这位学生这次考试的分数,因为找不到符合条件的记录去删除,这条DELETE语句也不会有什么危害。
- 如果这个考试分数不是一个空值,enter\_scores()函数将先对它进行一些基本的检查以保证它是一个数字。注意,这里的整数检查是通过模式匹配操作完成的,没有使用PHP的is\_int()函数。这是因为通过HTML表单返回的参数值都是被编码为字符串的,而is\_int()函数只能用来判断某个变量的类型是不是整数,它对任何字符串(哪怕它完全由数字字符构成)的判断结果都将是FALSE。我们这里需要的是一种能够判断字符串是否完全由数字字符构成的测试,模式匹配更能满足这一要求。如果字符串\$str完全由数字字符构成,下面这个测试将返回TRUE:

```
preg_match ('/^\\d+$/ ', $str)
```

如果考试分数通过了这个测试,就把它添加到score数据表里去。注意,用来完成这一步骤的数据库命令是REPLACE而不是INSERT,这是因为将要进行的数据库操作是替换一条现有记录而不是插入一条新记录。如果score数据表里原本没有这位学生这次考试的分数,REPLACE语句将插入一条新记录,就像执行了一条INSERT语句一样;否则,REPLACE语句就会用新记录替换掉旧记录。

到这里,score\_entry.php脚本就全部编写完了,可以用Web浏览器去录入或者编辑学生们的考试分数了。这个脚本有一个很明显的缺陷——没有实现任何安全措施,只要能连接到Web服务器,任何人都可以编辑学生们的考试分数。在后面的“美国历史研究会”会员资料在线修改脚本edit\_member.php里将实现一个简单的身份验证机制,大家可以把那个机制引入到这个脚本里来。如果数据的安全问题对你来说很重要,还可以考虑建立一个SSL连接来保护在Web浏览器与Web服务器之间传输的数据,但这些问题都超出了本书的讨论范围。

score\_entry.php脚本值得改进的地方还有:

- 检查有没有“坏”分数,比如某位学生没有参加考试却在score数据表里有一个分数记录或者两位学生的考试分数弄颠倒了等情况。
- 如果存在“坏”分数,在一个事务内录入考试分数并回滚事务。这一功能应该这样来实现:
  - 1) 把score数据表转换为一种能够支持事务处理的数据表类型,比如InnoDB;
  - 2) 在进入

用来完成考试分数录入工作的循环语句之前,先发出一条BEGIN语句;3)在循环语句里对用来表示事务操作是否成功的\$invalid\_count变量进行相应的设置;4)在循环语句的后面加上一段代码,让它根据\$invalid\_count变量的取值情况去执行一条COMMIT(事务成功)或ROLLBACK(事务不成功)语句。

如果决定修改这个脚本让它使用事务机制来完成有关工作的话,使用非永久性连接非常重要。一旦脚本在某次事务处理过程中意外“死亡”,你当然希望事务能够回滚,而这正是非永久性连接上将会发生的事情:PHP将关闭那条连接,MySQL则会在客户非正常退出执行的时候自动回滚尚未完成的事务。而如果使用的是永久性连接的话,因为PHP不会关闭那个连接,所以尚未完成的事务很可能无法得到回滚。

### 8.2.2 美国历史研究会:总统生平小测验

在“美国历史研究会”的会员刊物*Chronicles of U.S. Past*(美国历史年表)里有一个专为儿童准备的历史知识小测验栏目。我们打算把这个栏目办到研究会的Web站点上去。事实上,我们已经创建了president数据表,这样可以用它作为历史知识小测验的一个题目来源。下面编写一个pres\_quiz.php脚本来实现这个目标。

基本思路是这样的:随机挑选出一位总统,问一个与他有关的问题,获得用户给出的回答并检查这个回答是否正确。脚本给出的题目类型其实可以基于president数据表的任何一个部分,但为简单起见,我们将把它限制为只问总统们出生在什么地方。另一项简化措施是以选择题的形式来提出问题。这对用户比较方便,他们只需从一组选择里选出正确的答案就行了,用不着打字输入他们的回答。这对我们也很方便,因为我们不必为了检查用户敲入的答案而任何进行模式匹配,只需把用户做出的选择与正确答案比较一下就行了。

pres\_quiz.php脚本必须具备两个功能。第一,在第一次被调用的时候,它应该利用从president数据表查找出来的信息生成并显示一个新测验题。第二,如果用户提交了一个回答,脚本将检查它是否正确并给出相应的反馈提示。如果用户回答得不正确,脚本将重复显示刚才的问题;如果用户回答正确,脚本将生成并显示一个新问题。

pres\_quiz.php脚本的框架相当简单:如果用户没有提交一个回答,就显示初始的提问页面;如果用户提交了一个回答,就对它进行检查:

```
<?php
# pres_quiz.php - script to quiz user on presidential birthplaces

include "sampdb.php";

# ... put quiz-handling functions here ...

$title = "U.S. President Quiz";
html_begin ($title, $title);
sampdb_connect ()
    or die ("Sorry, could not connect to database; no quiz available");
```

```

$response = script_param ("response");
if (!isset ($response))      # invoked for first time
    present_question ();
else                          # user submitted response to form
    check_response ();

html_end ();
?>

```

我们将利用ORDER BY RAND()来随机生成小测验的题目,但这种用法只有MySQL 3.23.2及以后的版本才支持。利用RAND()函数,可以从president数据表里随机选取数据行。比如说,要想随机挑选出一个总统的姓名和出生地,只需发出下面这个查询就行了:

```

SELECT CONCAT(first_name, ' ', last_name) AS name,
CONCAT(city, ', ', state) AS place
FROM president ORDER BY RAND() LIMIT 1;

```

这个查询挑出来的姓名和出生地将是测验题“这位总统出生在哪一个城市?”的题目对象和正确答案。还得准备几个不正确的选择,可以用一个与上面类似的查询把它们选出来,如下所示:

```

SELECT DISTINCT CONCAT(city, ', ', state) AS place
FROM president ORDER BY RAND();

```

从这个查询的结果里,我们将选出与正确答案不同的前四项数据。在这个查询里使用DISTINCT关键字的原因是为了避免向用户提供的选择清单里出现重复的地名。要是美国总统们的出生地都不重复的话,当然不必使用DISTINCT关键字,可事实却并非如此,看看下面这个查询的结果就应该明白了:

```

mysql> SELECT city, state, COUNT(*) AS count FROM president
-> GROUP BY city, state HAVING count > 1;
+-----+-----+-----+
| city      | state | count |
+-----+-----+-----+
| Braintree | MA    | 2     |
+-----+-----+-----+

```

测验题和选择清单的生成工作由present\_question()函数负责,下面就是它的代码:

```

function present_question ()
{
    # issue query to pick a president and get birthplace
    $query = "SELECT CONCAT(first_name, ' ', last_name) AS name,"
        . " CONCAT(city, ', ', state) AS place"
        . " FROM president ORDER BY RAND() LIMIT 1";
    $result_id = mysql_query ($query)
        or die ("Cannot execute query");
    $row = mysql_fetch_array ($result_id)
        or die ("Cannot fetch result");
    $name = $row["name"];
}

```

```

$place = $row["place"];
# Construct the set of birthplace choices to present.
# Set up the $choices array containing five birthplaces, one
# of which is the correct response.
$query = "SELECT DISTINCT CONCAT(city, ', ', state) AS place"
        . " FROM president ORDER BY RAND()";
$result_id = mysql_query ($query)
        or die ("Cannot execute query");
$choices[] = $place;    # initialize array with correct choice
while (count ($choices) < 5 && $row = mysql_fetch_array ($result_id))
{
    if ($row["place"] == $place)
        continue;
    $choices[] = $row["place"]; # add another choice
}
# seed random number generator, randomize choices, then display form
srand ((float) microtime () * 10000000);
shuffle ($choices);
display_form ($name, $place, $choices);
}

```

如果使用的MySQL版本低于3.23.2, `present_question()`函数将无法工作, 因为老版本不允许在ORDER BY子句里使用函数。不过, 可以在sampdb发行版本的pres\_quiz.php脚本源代码中的注释里找到一些能够用来绕开这一限制的修改提示。

`present_question()`函数将调用一个名为`display_form()`函数来具体完成小测验题目信息的生成工作, 后者将在一个表单里显示一位总统的姓名、一组列出候选答案的单选按钮和一个Submit (提交) 按钮。这个表单最明显的用途是向用户显示测验题信息, 但它还需要做些其他的安排。它不仅要考虑到把测验题信息显示给用户, 还必须考虑到当用户提交了一个回答时, 怎样才能让脚本利用从Web返回的信息检查用户回答得是否正确并在用户回答错误时把刚才的小测验题目再次显示给用户。

把测验题信息显示给用户其实就是显示总统姓名和选择清单, 这件事很容易做到。但检查用户回答并在他们回答错误时重新显示测验题这件事就有点棘手了: 我们不仅要知道小测验题的正确答案, 还得知生成这道小测验题所必需的全部信息才行。要想达到这个目的, 一种办法是使用一组隐藏字段 (也叫做不可见字段) 把所有必要的信息都包容在表单里。这些隐藏字段将作为表单的组成部分在用户提交其小测验回答的时候发送回来, 但不会显示出来让用户们看到。

我们用了三个名为name、place和choices的隐藏字段来分别保存总统的姓名、正确出生地 (即小测验答案) 和提供给用户的选择清单。用`implode()`函数把选择清单里的各项选择编码为一个字符串。`implode()`函数能够方便地把多个字符串合并在一起并在各子串之间加上一个特殊的分隔符。( `explode()`函数则能够根据一个特殊的分隔符方便地把一个字符串拆分为多个子串, 在需要重新显示小测验题时会用到这个函数。) 下面就是负责具体生成这个表单的`display_form()`函数的代码:

```

function display_form ($name, $place, $choices)
{
    printf("<form method=\"POST\" action=\"%s\">\n", script_name ());
    hidden_field ("name", $name);
    hidden_field ("place", $place);
    hidden_field ("choices", implode ("#", $choices));
    printf ("Where was %s born?<br /><br />\n", htmlspecialchars ($name));
    for ($i = 0; $i < 5; $i++)
    {
        radio_button ("response", $choices[$i], $choices[$i], FALSE);
        print ("<br />\n");
    }
    print ("<br />\n");
    submit_button ("submit", "Submit");
    print ("</form>\n");
}

```

display\_form()使用了几个辅助函数来生成表单里的各种字段。第一个是hidden\_field()函数,它负责生成隐藏字段的<input>标记,下面就是它的代码:

```

function hidden_field ($name, $value)
{
    printf("<input type=\"%s\" name=\"%s\" value=\"%s\" />\n",
        "hidden",
        htmlspecialchars ($name),
        htmlspecialchars ($value));
}

```

很多脚本都可能会用到像hidden\_field()这样的通用例程,所以库文件sampdb.php应该是它的最佳去处了。另外,出于预防\$name和\$value变量的值里包含有特殊字符(比如引号)的考虑,它还使用PHP提供的htmlspecialchars()函数对<input>标记的name和value属性的值进行了编码。

另外两个辅助函数,即radio\_button()和submit\_button()的实现代码如下所示:

```

function radio_button ($name, $value, $label, $checked)
{
    printf("<input type=\"%s\" name=\"%s\" value=\"%s\" %s />%s\n",
        "radio",
        htmlspecialchars ($name),
        htmlspecialchars ($value),
        ($checked ? " checked=\"checked\"" : ""),
        htmlspecialchars ($label));
}

function submit_button ($name, $value)
{
    printf("<input type=\"%s\" name=\"%s\" value=\"%s\" />\n",
        "submit",
        htmlspecialchars ($name),
        htmlspecialchars ($value));
}

```



当用户从Web页面里选择了一个出生地并提交这个表单时,他选择的答案将作为response参数的值被发送回Web服务器,而我们只需在脚本里调用script\_param()函数就可以把这个参数值提取出来。pres\_quiz.php脚本还需要通过这个参数值来判断这是自己的首次调用(response参数没有值)还是后续调用(response参数有值,即用户提交的小测验答案)。换句话说,这个脚本将根据response参数是否有值来决定采取什么动作:

```
$response = script_param ("response");
if (!isset ($response))      # invoked for first time
    present_question ();
else                          # user submitted response to form
    check_response ();
```

我们还需要编写一个用来检查用户对小测验题的回答是否正确的check\_response()函数,这就要用到name、place和choices三个隐藏字段里的数据值了。我们已经知道,小测验的正确答案被编码在表单的place字段里,用户的回答保存在response参数里,只要把它们两个比较一下就能知道用户回答得是否正确了。根据这个比较的结果,check\_response()函数将先向用户提供一些反馈,然后再去生成一道新的测验题(用户回答正确)或者重新显示刚才的那道测验题(用户回答错误)。下面就是check\_response()函数的代码:

```
function check_response ()
{
    $name = script_param ("name");
    $place = script_param ("place");
    $choices = script_param ("choices");
    $response = script_param ("response");

    # Is the user's response the correct birthplace?

    if ($response == $place)
    {
        print ("That is correct!<br />\n");
        printf ("%s was born in %s.<br />\n",
            htmlspecialchars ($name),
            htmlspecialchars ($place));
        print ("Try the next question:<br /><br />\n");
        present_question();
    }
    else
    {
        printf ("%s is not correct. Please try again.<br /><br />\n",
            htmlspecialchars ($response));
        $choices = explode ("#", $choices);
        display_form ($name, $place, $choices);
    }
}
```

pres\_quiz.php脚本到这里就全部编写完了。现在,只需在“美国历史研究会”的主页上增加一个指向这个脚本的超链接,访问者们就能利用这个小测验来检查自己的知识水平了。

### 隐藏字段并不安全

为了在自己的前、后两次调用之间传递一些不想让用户们看到的信息，pres\_quiz.php脚本使用了几个隐藏字段。这种做法对于像这样的趣味性页面脚本还是很适用的。但是，如果真的有一些完全不想让用户看见的信息，就不应该使用隐藏字段来保存它们。因为隐藏字段毫无安全性可言。想知道为什么这样吗？把pres\_quiz.php脚本安装到Web服务器文档树下的ushl目录里，再从Web浏览器去请求它。等浏览器上出现小测验页面后，再通过浏览器的“View Source”（查看源代码）命令就能看到这个页面纯文本形式的HTML代码了。将会看到place隐藏字段的内容（也就是小测验题的正确答案）清清楚楚地显示在那里。说得再明白些，在总统生平小测验里作弊是很容易的。从pres\_quiz.php脚本的用途来讲，这种作弊手段并不构成多么严重的问题，但这个例子却证明了隐藏字段真的是毫无安全性可言。总而言之，如果不是想让用户看到的信息，就不应该把它们放在隐藏字段里。

### 8.2.3 美国历史研究会：会员个人资料的在线修改

我们将要编写的最后一个PHP脚本是edit\_member.php，它将使“美国历史研究会”的会员们能够通过网络以在线方式去编辑他们的会员名录资料。有了这个脚本，会员随时都可以去改正或者刷新他本人的会员资料而不必再向研究会递交申请报告。这一方面能使有关信息保持最新，同时也减少了你（研究会的秘书）的工作量。

很明显，我们必须做到会员资料只能由他本人或者你（研究会的秘书）来修改，这意味着必须增加一些安全措施。作为一种简单的身份验证机制的演示，我们将使用MySQL来存放各位会员的口令，会员只有在提供了正确的口令之后才能看到和使用这个脚本给出的记录修改表单。这个脚本的工作流程如下所示：

- 在首次调用时，edit\_member.php脚本将显示一个登录页面，会员必须在这个页面的表单里输入自己的会员ID编号和一个口令。
- 当会员提交登录表单时，脚本将根据口令数据表来检查会员输入的ID编号和口令是否正确。如果口令正确，脚本将从member数据表里把这位会员的个人资料检索出来并显示在一个HTML表单里供会员进行编辑。
- 当会员提交编辑过的表单时，我们将用这个表单里的内容更新数据库里的会员记录。

在开始上述工作之前，我们首先要把口令准备好。一个比较简便的办法是随机生成这些口令。下面两条MySQL语句将创建一个名为member\_pass的数据表并为每位会员分配一个口令，每个口令都是一个随机数的MD5校验和的前8个字符。出于演示的目的，这里编写的edit\_member.php脚本将采用这种简单而又快捷的办法，但在实际应用中，应该让会员自己去挑选一个口令：

```
mysql> CREATE TABLE member_pass (
    -> member_id INT UNSIGNED NOT NULL PRIMARY KEY,
    -> password CHAR(8));
```

```
mysql> INSERT INTO member_pass (member_id, password)
-> SELECT member_id, LEFT(MD5(RAND()), 8) AS password FROM member;
```

MD5()函数在MySQL 3.23.2之前的版本里没有提供。下面这条语句也能生成8个字符的随机值,而且适用于任何版本的MySQL:

```
mysql> INSERT INTO member_pass (member_id, password)
-> SELECT member_id, FLOOR(RAND()*99999999) AS password FROM member;
```

不过,这条语句生成的随机值不如那些基于MD5()函数的随机值变化多,因为它们全部由数字字符构成。

除了要为member数据表里列出的每一位会员生成一个口令外,我们还需要在member\_pass数据表里增加一个特殊的“0号会员”记录项,“0号会员”的口令相当于一位超级用户的口令。身为研究会的秘书,你必须知道这个超级口令才能去修改任何一位会员的个人资料:

```
mysql> INSERT INTO member_pass (member_id, password) VALUES(0, 'bigshot');
```

**注意** 在创建member\_pass数据表之前,应该把samp\_brows.pl脚本从Web服务器的脚本目录移出。(这个脚本是在第7章里编写的。如果被人利用,它能查看到sampdb数据库里任何一个数据表的内容——member\_pass数据表也不例外。因此,如果不想让别人有机会看到各位会员的口令或者超级口令的话,就应该把它转移出去。)

建立好member\_pass数据表之后,就可以开始编写edit\_member.php脚本了。下面是这个脚本的框架:

```
<?php
# edit_member.php - Edit Historical League member entries via the Web

include "sampdb.php";

# define action constants
define ("SHOW_INITIAL_PAGE", 0);
define ("DISPLAY_ENTRY", 1);
define ("UPDATE_ENTRY", 2);

# ... put input-handling functions here ...

$title = "U.S. Historical League -- Member Editing Form";
html_begin ($title, $title);

sampdb_connect ()
    or die ("Cannot connect to server");

# determine what action to perform (the default if
# none is specified is to present the initial page)

$action = script_param ("action");
if (!isset ($action))
```

```

$action = SHOW_INITIAL_PAGE;

switch ($action)
{
case SHOW_INITIAL_PAGE:      # present initial page
    display_login_page ();
    break;
case DISPLAY_ENTRY:          # display entry for editing
    display_entry ();
    break;
case UPDATE_ENTRY:           # store updated entry in database
    update_entry ();
    break;
default:
    die ("Unknown action code ($action)");
}

html_end ();
?>

```

初始页面由display\_login\_page()函数负责生成, 它将生成一个供会员们输入其ID 编号和口令的表单, 下面就是这个函数的代码:

```

function display_login_page ()
{
    printf ("<form method=\"POST\" action=\"%s?action=%s\">\n",
            script_name (),
            urlencode (DISPLAY_ENTRY));
    print ("Enter your membership ID number and password,\n");
    print ("then select Submit.\n<br /><br />\n");
    print ("<table>\n");
    print ("<tr>");
    print ("<td>Member ID</td><td>");
    text_field ("member_id", "", 10);
    print ("</td></tr>");
    print ("<tr>");
    print ("<td>Password</td><td>");
    password_field ("password", "", 10);
    print ("</td></tr>");
    print ("</table>\n");
    submit_button ("button", "Submit");
    print "</form>\n";
}

```

表单里的标签文字和输入框都被安排在一个HTML表格里, 这是为了让它们排列得整齐些。因为这里只涉及到两个字段, 所以效果似乎不太明显。其实, 这是一种非常有用的技巧, 尤其是在创建的表单里需要显示多个不同长度的字段的时候。只需把它们安排到一个HTML表格里, 它们就能上下排列得整整齐齐了, 从而便于用户阅读和理解。

display\_login\_page()函数还用到了两个辅助函数,它们都可以在sampdb.php库文件里找到。text\_field()负责生成一个文本输入框,下面就是这个函数的代码:

```
function text_field ($name, $value, $size)
{
    printf (<input type=\"%s\" name=\"%s\" value=\"%s\" size=\"%s\" />\n",
           "text",
           htmlspecialchars ($name),
           htmlspecialchars ($value),
           htmlspecialchars ($size));
}
```

除type属性的值是password以外, password\_field()函数的代码与上面完全一样,所以这里就不把它写出来了。

当用户在输入完会员ID编号和口令后提交这个表单时, action参数的值将是DISPLAY\_ENTRY, 所以edit\_member.php脚本中的switch语句将在脚本的下一次调用中执行display\_entry()函数。display\_entry()函数负责检查用户输入的口令是否正确, 如果正确, 就把会员的个人资料显示出来。下面是这个函数的代码:

```
function display_entry ()
{
    # Get script parameters; trim whitespace from ID, but
    # not from password, because password must match exactly.

    $member_id = trim (script_param ("member_id"));
    $password = script_param ("password");

    if (empty ($member_id))
        die ("No member ID was specified");
    if (!preg_match ('/^\\d+$/ ', $member_id))    # must look like integer
        die ("Invalid member ID was specified (must be an integer)");
    if (empty ($password))
        die ("No password was specified");
    if (check_pass ($member_id, $password)) # regular member
        $admin = FALSE;
    else if (check_pass (0, $password))    # administrator
        $admin = TRUE;
    else
        die ("Invalid password");

    $query = sprintf ("
        SELECT
            last_name, first_name, suffix, email, street, city,
            state, zip, phone, interests, member_id, expiration
        FROM member WHERE member_id = %s
        ORDER BY last_name
    ", quote_value ($member_id));
    $result_id = mysql_query ($query);
    if (!$result_id)
        die ("Cannot execute query");
}
```



```
if (mysql_num_rows ($result_id) == 0)
    die ("No user with member_id = $member_id was found");
if (mysql_num_rows ($result_id) > 1)
    die ("More than one user with member_id = $member_id was found");

printf ("<form method=\"POST\" action=\"%s?action=%s\">\n",
        script_name (),
        urlencode (UPDATE_ENTRY));

# Add member ID and password as hidden values so that next invocation
# of script can tell which record the form corresponds to and so that
# the user need not re-enter the password.

hidden_field ("member_id", $member_id);
hidden_field ("password", $password);

# Read results of query and format for editing

$row = mysql_fetch_array ($result_id);

print ("<table>\n");

# Display member ID as static text

display_column ("Member ID", $row, "member_id", FALSE);

# $admin is true if the user provided the administrative password,
# false otherwise. Administrative users can edit the expiration
# date, regular users cannot.

display_column ("Expiration", $row, "expiration", $admin);

# Display other values as editable text

display_column ("Last name", $row, "last_name");
display_column ("First name", $row, "first_name");
display_column ("Suffix", $row, "suffix");
display_column ("Email", $row, "email");
display_column ("Street", $row, "street");
display_column ("City", $row, "city");
display_column ("State", $row, "state");
display_column ("Zip", $row, "zip");
display_column ("Phone", $row, "phone");
display_column ("Interests", $row, "interests");

print ("</table>\n");

submit_button ("button", "Submit");
print "</form>\n";
}
```

display\_entry()函数做的第一件事是对口令进行验证。给定一个会员ID号, 如果用户给出的口令与这位会员在member\_pass数据表里的口令相匹配, 或者如果它与超级口令(即那位特殊的“0号会员”的口令)相匹配, edit\_member.php脚本就将把这位会员的个人资料显示在一个表单里供用户进行编辑。负责具体完成口令检查工作的check\_pass()函数将通过一个简单的数据库查询命令从member\_pass数据表里选出一条记录并把该记录中的password数据列的取值与用户在登录表单里输入的口令进行比较, 如下所示:

```
function check_pass ($id, $pass)
{
    $query = sprintf ("SELECT password FROM member_pass WHERE member_id = %s",
                      quote_value ($id));
    $result_id = mysql_query ($query);
    if (!$result_id)
        die ("Error reading password table");
    if ($row = mysql_fetch_array ($result_id))
        return ($row["password"] == $pass); # TRUE if password matches
    return (FALSE);                        # no record found
}
```

如果口令验证无误, display\_entry()函数将从member数据表里检索出这位会员的个人资料记录并用这条记录里的信息生成一个编辑表单。这个表单里的大多数字段都是允许用户进行编辑修改的文本输入框, 但这里有两个例外: 首先, member\_id值将被显示为静态文本, 因为它是我们用来惟一地确定某一位会员的键值, 所以不允许改变。第二, 会员资格的失效日期也不允许由会员本人进行修改——否则, 如果会员本人把这个日期修改为一个距今更远的日期的话, 其效果就将是他没有交纳会费却延续了自己的会员资格。但是, 如果用户在登录表单里给出了超级口令的话, 脚本就将把这个失效日期显示为可编辑字段。作为研究会的秘书, 你是知道这个口令的, 而你应该替那些交纳了会费的会员延续他们的会员资格失效日期。

字段标签和字段值的显示工作是由display\_column()函数负责完成的。这个函数的输入参数依次为: \$label, 文本输入框旁边的提示信息(即字段标签); \$row, 用来存放待编辑记录的数组; \$col\_name, 与该字段对应的数据列的名称, 用户输入到这个字段里的信息将被存放到member数据表的\$col\_name数据列里去; \$editable, 一个用来表明该字段是否允许修改的布尔值, display\_column()函数将根据这个变量把字段显示为可编辑字段或静态文本, 它的默认值是TRUE, 即允许修改。下面就是display\_column()函数的代码:

```
function display_column ($label, $row, $col_name, $editable = TRUE)
{
    print("<tr>\n");
    printf("<td>%s</td>\n", htmlspecialchars ($label));
    print("<td>");
    if ($editable) # display as edit field
        text_field ("row[$col_name]", $row[$col_name], 80);
    else          # display as read-only text
        print (htmlspecialchars ($row[$col_name]));
    print("</td>\n");
    print("</tr>\n");
}
```

对于可编辑值, `display_column()` 函数将以具有 `row[col_name]` 格式的名字生成一个文本框。这样, 等用户提交这个表单的时候, PHP 就能把这些字段的值放到一个数组变量里去, 而这个数组里各个元素的键值就是它所对应的数据列名称 (`col_name`)。这一方面能让我们很容易地把表单的内容提取出来; 另一方面, 当我们准备更新数据库里的记录时, 能让我们方便地把表单里的各个字段与 `member` 数据表里的各个数据列对应起来。比如说, 假设已经把对应于某位会员的参数数组提取到了变量 `$row` 里, 那么这位会员的电话号码就保存在数组元素 `$row["phone"]` 里。

`display_entry()` 函数还通过两个隐藏字段把 `member_id` 和 `password` 参数的值嵌在了它生成的表单里, 这样, 当用户提交这个表单的时候, 我们就能把这两个参数值传递到 `edit_member.php` 脚本的下一次调用里去了。`member_id` 参数给出的会员 ID 编号将告诉脚本需要对 `member` 数据表里的哪一条记录进行更新, `password` 参数给出的口令将使脚本能够再次对用户的身份进行验证以确保他就是刚才登录的那位会员。(注意, 我们实现的这个简单的身份验证机制将以明文的形式在 Web 服务器和 Web 浏览器之间来回传递口令, 这通常不是一个好的解决方案。幸好“美国历史研究会”并不是一个对安全性要求很高的机构, 这个方案已经足以满足要求。如果处理的是一些金融数据, 就应该使用一种更安全的身份验证机制。)

用户提交回来的编辑表单将由 `edit_member.php` 脚本中的 `update_entry()` 函数负责处理, 它将负责具体完成会员资料在数据库里的更新工作。下面就是这个函数的代码:

```
function update_entry ()
{
    # Get script parameters; trim whitespace from ID, but
    # not from password, because it must match exactly, or
    # from row, because it is an array.

    $member_id = trim (script_param ("member_id"));
    $password = script_param ("password");
    $row = script_param ("row");

    $member_id = trim ($member_id);
    if (empty ($member_id))
        die ("No member ID was specified");
    if (!preg_match ('/^\\d+$/ ', $member_id))      # must look like integer
        die ("Invalid member ID was specified (must be an integer)");
    if (!check_pass ($member_id, $password) && !check_pass (0, $password))
        die ("Invalid password");

    # We'll need a result set to use for assessing nullability of
    # member table columns. The following query provides one without
    # selecting any rows. Use the query result to construct an
    # associative array that maps column names to true/false values
    # indicating whether columns allow NULL values.

    $result_id = mysql_query ("SELECT * FROM member WHERE 1 = 0");
    if (!$result_id)
        die ("Cannot query member table");
```

```

$nullable = array ();
for ($i = 0; $i < mysql_num_fields ($result_id); $i++)
{
    $fld = mysql_fetch_field ($result_id, $i);
    $nullable[$fld->name] = !$fld->not_null;    # TRUE if nullable
}
mysql_free_result ($result_id);

# Iterate through each field in the form, using the values to
# construct the UPDATE statement.

$query = "UPDATE member ";
$delim = "SET";
reset ($row);
while (list ($col_name, $val) = each ($row))
{
    $query .= "$delim $col_name=";
    $delim = ",";
    # if a form value is empty, update the corresponding column value
    # with NULL if the column is nullable. This prevents trying to
    # put an empty string into the expiration date column when it
    # should be NULL, for example.
    $val = trim ($val);
    if (empty ($val))
    {
        if ($nullable[$col_name])
            $query .= "NULL";    # enter NULL
        else
            $query .= "\'\'";    # enter empty string
    }
    else
        $query .= quote_value ($val);
}
$query .= sprintf (" WHERE member_id = %s", quote_value ($member_id));
if (mysql_query ($query))
    print ("Member entry was updated successfully.\n");
else
    print ("Member entry was not updated.\n");
}

```

这个函数先要再次检查口令以预防这是一份别人发来欺骗我们的假表单,然后再对会员记录进行更新。这个更新操作需要一些前期处理:如果用户提交回来的编辑表单里有空白字段(即用户没有在该字段填入相应的信息),可能需要把这些空字符串转换为NULL值之后再插入到MySQL数据库里去。expiration数据列就是一个这样的例子。比如说,假设研究会现在决定接纳某位会员为终身会员,身为研究会秘书的你就需要以超级口令登录进入edit\_member.php脚本所生成的编辑表单并把该会员的会员资格失效日期字段清除为空白。在MySQL数据库里,终身会

员是以其会员资格失效日期被设置为NULL值为标志的，所以我们必须先把从会员记录编辑表单的会员资格失效日期字段返回的空字符串转换为NULL值，然后再去更新这位会员在member数据表里的记录。如果edit\_member.php脚本在用户提交会员记录编辑表单的时候没有进行这种转换就直接把一个空字符串插入到了member数据表的expiration数据列里，MySQL将把它转换为一个'0000-00-00'形式的日期值，很明显这是一个错误。因此，当提交回来的会员记录编辑表单里有空白字段时，我们首先要判断与之对应的MySQL数据列是否需要先把这个空字符串转换为NULL值并做好相应的处理，然后再去更新有关的记录。

我们是这样来解决这一问题的。先让update\_entry()函数去检索member数据表的元数据并构造出一个这样的数组：各数组元素的键值是member数据表里各数据列的名字，各数组元素的值则表明了该数据列是否允许使用NULL值。这些信息是由mysql\_fetch\_field()函数返回的。当我们需要检查某个数据表的各个数据列是否允许使用NULL值时，先对这个数据表发出一条数据库查询命令，然后再把这个查询命令的结果集标识符传递给mysql\_fetch\_field()函数，就能得到我们需要的信息了。在edit\_member.php脚本的update\_entry()函数里，我们使用了一条下面这样的SELECT语句：

```
SELECT * FROM member WHERE 1 = 0
```

这条查询命令返回的是一个空结果集，但这并不影响我们想要达到的目的，因为我们只对这个结果集的标识符感兴趣。只要把这个结果集标识符传递给mysql\_fetch\_field()函数，我们就能够获得所需要的信息——关于member数据表里的各个数据列是否允许使用NULL值的元数据。

到这里，edit\_member.php脚本就全部编写完成了。把这个脚本安装到Web服务器文档树的ushl目录，再让会员们知道他们各自的口令，他们就能通过Web以在线方式来修改他们自己的个人资料了。





## 第三部分 MySQL系统管理

## 第9章 MySQL系统管理简介

## 第10章 MySQL的数据目录

## 第11章 MySQL 数据库系统的日常管理

## 第12章 MySQL安全技术

## 第13章 MySQL数据库的备份、维护和修复

壯觀奇麗野營 1.8

了会本补，益用自谷国守叶判里些互下悉燕育只。知辨料这个凡我由楚系率谢媛JQ2yM  
并逐。责即取管即若如宗与自胡得求乳辟具工韵御玉用数念本，因本位补工照管楚系率谢媛种  
儿不以的JQ2yM成燕亥巡梯，拙试。手宜心样更雅来站潜工，西来的慨海消甚思去同担些

面式个

MySQL数据库的官方文档中，对MySQL数据库的架构进行了详细的描述。MySQL数据库的架构可以分为以下几个层次：

数据库设计工具。数据库设计工具是数据库设计过程中不可或缺的一部分。数据库设计工具可以帮助数据库设计人员快速、准确地完成数据库设计工作。数据库设计工具可以分为以下几类：

## 第9章 MySQL系统管理简介

在各种数据库系统中，MySQL是比較容易使用的，它的安装工作也不复杂。MySQL之所以会受到人们——尤其是那些不是也不想成为数据库管理员的人们——的欢迎，主要原因就是它的这种易用性。即使不是计算机专家，你也能成功地运行一个MySQL数据库系统；如果你是，那当然就更好了。

不管你是不是计算机专家，MySQL肯定不会自己运转起来——必须有人去启动它。为了让它运行得既平稳又有效率，就要有人去管理它，就要有人在它出问题时知道该怎么办。如果这个任务落在了你的头上，请继续往下读。

本书的第三部分将向大家介绍MySQL管理工作的各个方面。这一章是对MySQL管理工作的概括性介绍，后面几章是对各项职责的具体论述。

如果你是一位初出茅庐的MySQL数据库管理员，请不要因为本章所列的一长串职责清单而打退堂鼓。这些职责中的每一项都很重要，但并不需要把它们一次都学会，你完全可以根据自己遇到的问题来随时查阅本书这一部分的有关章节。

如果你有管理其他数据库系统的经验，就会发现MySQL与它们有着很多共同之处，你的经验将是一笔宝贵的财富。但MySQL的管理工作还有一些独特的要求，而本书的这一部分将帮助你熟悉这些要求。

### 9.1 管理职责概述

MySQL数据库系统由好几个组件构成。只有熟悉了这些组件和它们各自的用途，你才会了解数据库系统管理工作的本质，才会选用正确的工具程序来帮助自己完成各项管理职责。多花些时间去思考你读到的东西，工作起来就能更加得心应手。为此，你应该熟知MySQL的以下几个方面：

- **MySQL服务器。**服务器程序mysqld是整个MySQL数据库系统的核心；所有的数据库和数据表操作都是由它来完成的。mysqld\_safe是一个用来启动、监控和（在出问题时）重新启动mysqld的相关程序。（mysqld\_safe在MySQL 4之前的版本里叫做safe\_mysqld）。如果在同一台主机上运行了多个服务器，就需要用mysqld\_multi程序来帮助自己管好它们。
- **MySQL客户程序和工具程序。**有几个MySQL程序是用来与服务器进行通信的。就管理工作而言，最重要的是下面这几个：
  - mysql——这是一个用来把SQL语句发往服务器并让你查看其结果的交互式程序。
  - mysqladmin——这是一个用来完成关闭服务器或在服务器运行不正常时检查其运行状态等工作的管理性程序。
  - mysqlcheck、isamchk和myisamchk——这几个工具用来对数据表进行分析和优化。当数据表损坏时，还可以用它们来进行崩溃恢复工作。

- **mysqldump和mysqlhotcopy**——用来备份数据库或者把数据库拷贝到另一个服务器的工具。
- **服务器的语言，SQL**。有些管理工作可以只使用命令行工具程序mysqladmin完成，但如果能用服务器自己的语言与它“交谈”则更好。比如说，如果想知道你为用户设置的权限为什么没有起到应有的作用，直接与服务器进行对话——用mysql客户程序发出SQL查询命令去检查权限表当然是最好的办法。如果你的MySQL版本没有提供GRANT语句，mysql程序还允许用直接修改权限表的办法设置各个用户的权限。  
如果从未接触过SQL，你至少应该了解它的基本概念。缺乏SQL知识将妨碍你自己的工作，而花在学习SQL上的时间将给你带来更多的回报。想精通SQL当然要花不少时间，但其基本技能却很容易掌握。关于SQL语句及命令行客户程序mysql使用方法的介绍请参阅本书的第1章。
- **MySQL数据目录**。数据目录是服务器用来保存数据库以及各种状态文件的地方。只有熟悉了数据目录的结构与内容，你才能明白服务器是如何用文件系统来表示数据库和数据表的，才能知道各种状态文件（如日志文件）的存放位置及其内容，才能在数据目录所在的文件系统空间不足时用最适当的手段去调整磁盘空间。

## 9.2 日常管理

日常管理的主要职责是对MySQL服务器程序mysqld的运行情况进行管理，使数据库用户能够顺利地访问MySQL服务器。下面是这项工作的主要职责：

- **服务器的启动与关闭**。这一职责的具体内容包括：1) 从命令行以手动方式启动和关闭MySQL服务器；2) 安排MySQL服务器在系统的开机和关机过程中自动地启动和关闭；3) 在MySQL服务器崩溃或者非正常启动时把它恢复到正常的运行状态。
- **对用户账户进行管理**。这一职责的具体内容包括：1) 了解MySQL用户账户与UNIX或Windows注册账户之间的区别；(2) 设置MySQL用户账户，限制用户只能从指定的机器上去连接MySQL服务器；3) 把正确的连接参数通知给新用户，使他们能顺利地连接上MySQL服务器——他们的工作是使用数据库而不是设置账户！4) 如果用户（或者你本人）忘记了口令，你还要知道怎样才能重新设置一个新口令。
- **对日志文件进行管理**。这一职责的具体内容包括：1) 知道自己都能对哪些类型的日志文件进行管理；2) 在什么时候以及如何去进行管理；3) 制定并实施日志循环和失效机制，防止日志文件把文件系统的可用空间消耗殆尽。
- **对数据库进行备份和搬迁**。当系统发生崩溃的时候，数据库备份将发挥至关重要的作用。你肯定希望自己能够以尽可能小的数据损失把数据库恢复到崩溃发生之前的状态。但要注意的是，数据库备份工作与普通意义上的系统备份工作（比如用UNIX工具程序dump进行的备份工作）是有区别的。系统备份工作通常由系统管理员负责，他在备份工作开始之前不一定把MySQL服务器关闭掉。于是，在系统备份工作的进行过程中，可能会有某些数据表的内容因为MySQL服务器仍在对它们进行着读写而发生变化——用这样的备份来恢



复系统将导致那些数据表的内容发生错乱。mysqldump程序生成的备份文件更适用于数据库恢复操作，而且它不要求你必须在备份工作开始之前先关闭MySQL服务器。你还可能需要在磁盘满时移动数据库。

数据库的搬迁指的是把数据库从一个硬盘转移到另一个硬盘上去。当磁盘的可用空间所剩无几时或者当你想把某些数据库转移到另一台速度更快的主机上时，就需要对有关的数据库进行搬迁。这里要提醒大家注意这样一个问题：数据库文件依赖于具体的操作系统，所以数据库的搬迁操作不一定总能用简单的文件拷贝命令来完成。

- **建立数据库镜像。**如果把对数据库进行备份或者拷贝比喻为给数据库拍“照片”的话，建立数据库镜像就相当于给数据库拍“录像”了。建立数据库镜像需要同时运行两个数据库服务器并使它们构成主、从关系，这样对主服务器所管理的某个数据库所做的修改将同步地（可能稍有延迟）反映在从服务器所管理的与之对应的数据库里。
- **对服务器进行配置和优化。**数据库用户都希望数据库服务器运行在最佳状态，而改善服务器性能的最简单方法是添置更多的内存和更高速的硬盘。但这绝不能成为你不钻研数据库工作原理的理由——在如此“蛮干”之后，仍需要对服务器进行配置和优化。这一职责的具体内容包括：1）知道有哪些参数可以用来对服务器进行优化；2）如何根据具体情况来进行这些优化。某些站点上的查询多为数据检索操作，而另一些站点上的查询却多为数据插入和修改操作。具体到你的站点，应该根据实际观察到的查询“混合比例”去选择最有效的参数来加以改变。

对数据库服务器进行“本地化”（比如设置适当的字符集和时区等）也是其配置工作之一。

- **同时运行多个服务器。**某些场合需要同时运行多个服务器。你或许是想对MySQL软件的一个新版本进行测试但又必须让现有的服务器保持运行，或许是想通过让不同的用户组去使用不同的服务器以便为各组用户提供更好的隐私保护机制。（后一种情况特别适用于ISP。）无论哪一种情况都需要你掌握同时安装并启动多个MySQL服务器的技术。
- **对MySQL软件进行升级。**与其他软件产品一样，MySQL也在不断地更新换代。想使用漏洞更少、功能却更丰富的新版本，就必须掌握软件的升级技术。这一职责的具体内容包括：1）知道如何对MySQL软件进行升级；2）在哪些情况下不进行升级更合理；3）如何在稳定版本和测试版本之间做出选择。

### 9.3 安全问题

无论由谁来负责MySQL服务器的管理工作，都必须确保用户存放在数据库里的信息是安全的。对数据目录和服务器的访问情况进行控制是MySQL数据库管理员的重要职责之一。为了做好这项工作，必须知道如何完成以下工作：

- **加强文件系统的安全性。**UNIX计算机往往有多个系统管理员账户，但对MySQL服务器进行管理却不是所有这些账户的职责之一。必须确保与MySQL服务器管理工作无关的UNIX系统管理员账户不具备访问MySQL数据目录的权限，让它们既不能拷贝或删除数据库文件，也不能读取MySQL日志文件里的敏感信息。这样，别人就无法在文件系统层次上盗取或者破坏数据库里的数据了。这一职责的具体内容包括：1）建立一个专门用来运行

MySQL服务器的UNIX用户账户；2) 把该用户设置为MySQL数据目录的属主；3) 启动MySQL服务器运行在该用户的权限范围内。

- **加强MySQL服务器的安全性。**这一职责的具体内容包括：1) 了解MySQL安全系统的工作原理；2) 为新建立的用户账户分配适当的权限。必须特别留意那些通过网络来连接MySQL服务器的账户，分配给这类账户的权限绝不能超出它们在正常使用时所必需的。如果你对MySQL安全系统的工作原理不了解，就难免失手把超级用户的权限授予给匿名用户——而这是绝对不允许的！

## 9.4 数据库修复和维护

MySQL数据库管理员都不希望遇上错误百出或者被人为破坏的数据表，但希望不能代替现实。必须采取措施降低这种风险，同时还要知道在意外发生时应该如何去应对，所采取的措施包括：

- **崩溃恢复。**如果你已经尽了最大努力但意外仍然发生了，必须知道如何去修复或者用备份去恢复数据表。崩溃恢复事件极少发生，可一旦发生，就意味着一段痛苦而又高度紧张的日子（尤其是在你正忙得不可开交而电话也响、门铃也响的时候）。可话又说回来了，谁让你是数据库管理员呢？想让用户满意，就必须把事情做好。这一职责的具体内容包括：1) 熟练掌握对MySQL数据表进行检查和修复的工具程序；2) 知道如何利用备份文件来恢复受损数据；3) 知道如何利用MySQL变更日志来恢复最近一次备份后又发生的数据修改操作。
- **预防性维护。**为了降低数据库出现故障或被破坏的可能性，应该提前制定出一份预防性的维护制度。备份工作也应该制度化，但预防性维护可以减少使用备份的机会。

以上就是作为MySQL数据库管理员所必须承担的职责。在接下来的几章里，将对各项职责及其实施步骤做详细的讨论以帮助大家有效地履行这些职责。我们的讨论将从MySQL数据目录开始（因为各种数据库管理工作几乎都是在这项资源上进行的，必须了解它的布局和内容），然后再逐步深入到MySQL数据库系统的日常管理职责、MySQL数据库系统的安全系统以及MySQL数据库的维护和故障排除。



## 第10章 MySQL的数据目录

从概念上讲,大多数关系数据库管理系统都是相似的:它们管理着一组数据库,每个数据库又包含着一组数据表。但它们在数据的组织和管理方面各有特点,MySQL也不例外。在默认情况下,由MySQL服务器程序mysqld负责管理的所有信息都存放在一个人们称之为“MySQL数据目录”的地方——数据库全都存放在那里,用来提供服务器运转情况信息的状态文件和日志文件也都存放在那里。如果你承担着MySQL数据库管理工作的某项职责,熟悉MySQL数据目录的布局及用途将是开展工作的先决条件。即使你不承担任何MySQL管理职责,阅读本章也能获得好处;多掌握些关于MySQL服务器工作原理方面的知识绝不会有坏处。

本章将对以下几个问题进行讨论:

- **如何确定MySQL数据目录的位置。**想有效地对数据库进行管理,就必须知道这个位置。
- **服务器是如何组织和管理数据库与数据表的,又是如何把它们展现在用户面前的。**这对预防性维护制度的制定工作以及受损数据表的崩溃恢复工作有着重要的意义。
- **服务器会生成哪些状态文件和日志文件以及这些文件的内容。**这些文件的内容提供了有用的服务器工作情况信息。当你遇到问题时,这些信息的价值就体现出来了。
- **如何改变MySQL数据目录的默认位置或布局。**这对系统上的硬盘资源分配工作——比如当需要把硬盘活动分散到不同的磁盘或者需要把数据搬迁到一个有着更多可用空间的文件系统上时——有着重要的意义。还可以利用这部分知识去为新数据库安排一个适当的存放地点。

对于UNIX系统,本章将假设你已经创建了一个专门用来执行MySQL数据库管理工作和运行服务器的登录账户,而这个登录账户的用户名和用户组名分别是mysqladm和mysqlgrp。使用一个专用的登录账户进行MySQL管理工作的理由将在第11章中讨论。

### 10.1 数据目录的位置

MySQL数据目录的默认位置已经被编译到MySQL服务器程序里了。在UNIX系统上,这个默认位置通常是/usr/local/mysql/var(如果是用一个源代码发行版本来安装MySQL的话)、/usr/local/mysql/data(如果是用一个二进制发行版本来安装MySQL的话)或/var/lib/mysql(如果是用一个RPM文件安装MySQL的话)。在Windows系统上,默认的数据目录是C:\mysql\data。

在启动MySQL服务器的时候,可以用--datadir=dir\_name选项来明确地为MySQL数据目录指定一个位置。这在需要把MySQL数据目录安排到默认位置以外的某个地方时很有用。把MySQL数据目录安排到其他地方的另一种办法是把它列在MySQL服务器在启动时会读取的某个选项文件里。这样,就不必在每次启动MySQL服务器时都在命令行上写出其数据目录的路径了。为MySQL数据目录另行指定一个位置的具体步骤将在本章后面的内容里介绍。

作为MySQL数据库管理员，必须知道服务器的数据目录在什么地方。如果同时运行了多个服务器，就必须知道它们各自的数据目录在什么地方。如果不知道这个位置（比如你的前任没有给你留下这方面线索的时候），可以用下面几种办法把它查出来：

- 向服务器询问这个位置。MySQL服务器维护着很多与其运转情况有关的变量并能够把这些变量的值报告给你。MySQL数据目录对应着datadir变量，它的值可以用mysqladmin variables命令或SHOW VARIABLES语句查出来。mysqladmin是一个在命令行上使用的工具程序，它在UNIX系统上的输出通常是如下所示的样子：

```
% mysqladmin variables
+-----+-----+
| Variable_name | Value |
+-----+-----+
...
| datadir       | /usr/local/mysql/var/ |
...
```

在Windows系统上，它的输出通常是如下所示的样子：

```
C:\> mysqladmin variables
+-----+-----+
| Variable_name | Value |
+-----+-----+
...
| datadir       | c:\mysql\data\ |
...
```

在mysql程序里，可以像下面这样来查知这个变量值：

```
mysql> SHOW VARIABLES LIKE 'datadir';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| datadir       | /usr/local/mysql/var/ |
+-----+-----+
```

如果同时运行着多个服务器，应该在不同的TCP/IP端口、套接字或命名管道上监听。如果你想查看在某特定端口或套接字上监听着的MySQL服务器的数据目录信息，只需使用相应的--port或者--socket选项连接到那个端口或套接字即可。明确地连接到127.0.0.1位置上的主机将让mysqladmin使用一条TCP/IP连接去连接运行在本地主机上的服务器，如下所示：

```
% mysqladmin --host=127.0.0.1 --port=port_num variables
```

在UNIX系统上，如果给出的--host选项值是localhost，系统将使用一个UNIX套接字去连接MySQL服务器。如果你是通过一个套接字文件去连接本地主机上的MySQL服务器的，请使用--socket选项，如下所示：

```
% mysqladmin --host=localhost --socket=/path/to/socket variables
```

在基于Windows NT的系统上，可以使用“.”作为一条命名管道连接的主机名，也可用--socket选项来给出命名管道的名字，如下所示：

```
C:\> mysqladmin --host=. --socket=pipe_name variables
```

无论何种平台，都可以通过--host选项来给出服务器主机名的办法来使用一条TCP/IP连接去连接到运行在另一台主机上的远程MySQL服务器，如下所示：

```
% mysqladmin --host=host_name variables
```

当需要连接到某个非默认端口时，还需要使用相应的--port选项给出具体的端口号。

- 在UNIX系统上，可以用ps命令查看当前执行的mysqld进程或其他进程的命令行。通过查找有关的--datadir选项值，能确定MySQL数据目录的位置。比如说，如果使用的是BSD风格的ps命令，不妨试试下面这条命令：

```
% ps axww | grep mysqld
```

如果使用的是System V风格的ps命令，请试试下面这条命令：

```
% ps -ef | grep mysqld
```

如果系统里运行有多个MySQL服务器，ps命令将特别有用，因为能一次查明多个MySQL数据目录的位置。这个办法的缺点是只能在服务器主机上执行ps命令，而且除非在mysqld命令行上明确地给出了--datadir选项，否则ps命令的输出里就不会有你想要的信息。（但从另一方面讲，某些会调用mysqld程序的启动脚本会尝试确定MySQL数据目录的路径名并把它放到mysqld命令行上，使ps命令能够查出这项信息。）

- 查看服务器在启动时所读取的选项文件，比如说，如果查看UNIX系统上的/etc/my.cnf文件或者Windows系统上的C:\my.cnf文件，通常可以在[mysqld]选项组里看到一个下面这样的datadir行：

```
[mysqld]
datadir=/path/to/data/directory
```

这里给出的路径名就是MySQL数据目录的位置。

- MySQL服务器的帮助信息里有一个在该服务器被编译时确定的数据目录默认位置。如果你在启动MySQL服务器时没有用另外一个路径来覆盖它的话，这个默认位置就应该是这个服务器在运行时实际使用的MySQL数据目录位置。下面这条命令可以查出MySQL数据目录的默认位置：

```
% mysqld --help
...
datadir      /usr/local/mysql/var/
...
```

- 如果MySQL软件是用某个源代码发行版本安装的，还可以通过它的配置信息来查明MySQL数据目录的位置。一般说来，这个位置可在最顶层的Makefile文件里查到。但要注意的是，在Makefile文件里，MySQL数据目录的位置是由localstatedir变量而不是datadir变量给出的。此外，如果源代码发行版本是被保存在一个NFS文件系统上并被用来在好几

个主机上建立过MySQL服务器，Makefile文件里的配置信息将只精确对应于最后建立的那个MySQL服务器，这可能不是你感兴趣的那台主机上的MySQL服务器所使用的数据目录位置。

- 如果上面的方法都失败了，还可以通过使用find命令搜索数据库文件的办法来查明MySQL服务器的数据目录位置。下面这条命令将寻找.frm（数据库定义）文件：

```
% find / -name "*.frm" -print
```

在.frm文件里存放着某MySQL服务器所管理的数据表的定义，也就是说，.frm文件肯定是某个MySQL安装的组成部分。这些.frm文件通常都有一个相同的父目录，而这个父目录就应该是MySQL服务器的数据目录。

在本章后续内容里的操作示例中，将使用DATADIR来代表MySQL数据目录的位置，读者应该把操作示例中的DATADIR解释为运行在自己机器上的MySQL服务器所使用的MySQL数据目录的位置。

## 10.2 数据目录的结构

MySQL数据目录收录着MySQL服务器所管理的全部数据库和数据表。一般来讲，它们将构成一个树状结构，这个树状结构是直接利用UNIX或Windows文件系统的层次结构而实现的，即：

- 每个数据库对应于MySQL数据目录下的一个目录。
- 同一数据库里的数据表对应于数据库目录中的各有关文件。

这种以目录和文件来实现数据库和数据表的层次化的做法有一个例外——InnoDB数据表处理程序把所有数据库里的所有InnoDB数据表全都存放在同一个公共表空间里。这个表空间是用一个或者多个非常大的文件而实现的，这些文件将被视为一个连续统一的数据结构，各InnoDB数据表的数据和索引都将存放在这个连续统一的数据结构中。在默认的情况下，InnoDB表空间文件也都存放在MySQL数据目录里。

MySQL数据目录还可能包含有其他一些文件，比如：

- MySQL服务器的选项文件my.cnf。
- MySQL服务器的进程ID（PID）文件。在启动时，MySQL服务器将把自己的进程ID写入这个文件，这样，当其他程序需要向MySQL发送信号时，就可以从这个文件里查知MySQL服务器的进程ID了。（注意：这个进程ID文件在基于Windows的系统或嵌入式MySQL服务器里不存在。）
- MySQL服务器所生成的状态和日志文件。这些文件里记录着关于MySQL服务器操作运行情况的重要信息，这些信息对系统管理员——尤其是在系统出现问题而又想查明问题的原因时——有着巨大的价值。比如说，如果MySQL服务器在执行某个查询时意外地死机了，往往可以通过日志文件来查知到底是哪一条查询命令导致的这次死机。
- 把DES密钥文件或服务器的SSL证书与密钥文件存放在MySQL数据目录里也是常见的做法。

## 10.2.1 MySQL服务器如何提供对数据的访问

当MySQL在普通的“客户/服务器”模式下启动时，相应的MySQL数据目录下的所有数据库都将由同一个实体——MySQL服务器程序mysqld——负责管理。各种客户程序从不直接对数据进行操作，它们必须通过服务器才能存取数据库里的数据；而服务器则是客户程序与它们所要使用的数据之间的惟一通路。图10-1给出了这种体系结构。

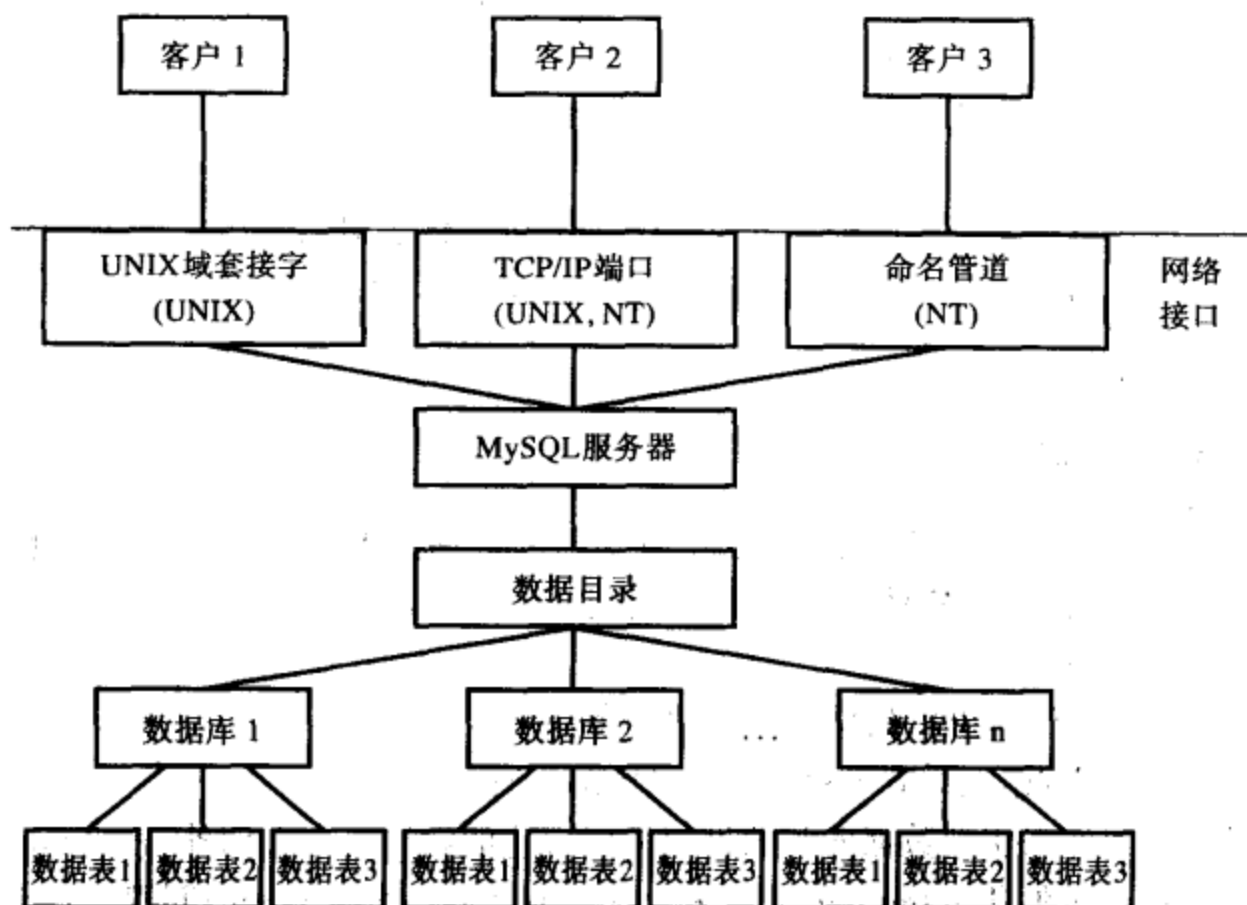


图10-1 MySQL服务器是如何控制对MySQL数据目录的存取访问的

在启动时，MySQL服务器将根据命令行（或选项文件）的要求打开一些日志文件，然后向用户提供一个到有关的MySQL数据目录的网络接口并开始在这个接口上监听各种网络连接。（服务器如何进行网络监听的详细情况见第11章。）为了访问数据，客户程序先要建立一个到服务器的连接，然后发出各种SQL查询命令让MySQL服务器执行有关的操作——比如创建一个数据表、选取数据记录、修改数据记录等等。服务器在执行完有关操作后把结果返回给客户。MySQL服务器程序是多线程的，能够同时向多个客户连接提供服务。不过，因为MySQL服务器每次只能执行一个修改操作，所以实际效果是有关请求将被“串行化”，即任意两个客户决不会在同一时刻修改某给定记录。

如果你运行的是一个使用了嵌入式MySQL服务器的应用软件，系统就将使用一个稍微不同的体系结构，这是因为此时仅有一个“客户”——即链接有MySQL服务器的应用程序。此时，MySQL服务器将监听一个内部通信通道而不是某个网络接口。但即使在这种情况下，对MySQL数据目录的访问操作也仍必须通过嵌入在该应用软件里的MySQL服务器部分去执行；如果这个应用软件打开了到MySQL服务器的多条连接，它就要对经由多条连接而到达的查询动作进行协调。



在正常情况下，让MySQL服务器充当数据库访问操作的惟一中转环节是有好处的，这能有效地避免因允许多个进程在同一时间访问同一个数据表而可能导致的数据损坏现象。然而，系统管理员必须知道MySQL服务器在下面几种场合里是无法独占性地控制MySQL数据目录的：

- **当在同一个数据目录上运行多个MySQL服务器时。**一般说来，人们只会运行一个MySQL服务器来管理同一台主机上的所有数据库，但偶尔也会出现在同一台主机上运行了多个MySQL服务器的情况。如果这些服务器都有它们各自专用的数据目录，就不会出现彼此干扰的问题。不过，启动多个MySQL服务器并让它们都指向同一个数据目录的情况也并非不可能发生——这通常不是个好主意。如果你想这么试试，请务必确认系统具备良好的文件级锁定机制，否则，多个MySQL服务器就很难得到协调。如果允许多个MySQL服务器同时往同一组日志文件里写东西，这些日志文件就会变成一个混乱之源（而不是一个帮助信息之源）。
- **当运行各种数据表修复工具程序时。**有些工具程序（比如用来对数据表进行维护、故障排除和修复的myisamchk和isamchk等）是直接在有关的数据表文件上进行操作的。因为它们会改变数据表的内容，所以如果在MySQL服务器正对某数据表进行存取的同时使用这些工具程序去修改那个数据表，就可能导致数据表被损坏。要想避免这类问题，最好的办法是在运行任何一种数据表修复工具程序之前关闭MySQL服务器。如果做不到这一点，至少也要保证MySQL服务器不会在你使用某个数据表修复工具程序的时候去访问有关的数据表。这些工具程序的正确用法请参见第13章中的有关内容。（另一种办法是通过CHECK TABLE和REPAIR TABLE语句让MySQL服务器去执行数据表维护操作——这样就不会产生访问冲突了。）

### 10.2.2 MySQL数据库在文件系统里如何表示

MySQL服务器所管理的每一个数据库都有它自己的数据库目录，这个数据库目录其实是MySQL数据目录中的一个子目录，这个子目录的名字与它代表的数据库名字相同。比如说，对应于数据库mydb的数据库目录就是DATADIR/mydb。这种表示方法使MySQL数据库系统中一些与数据库有关的语句实现起来相当简单。

比如说，SHOW DATABASES命令其实就相当于列出MySQL数据目录中的目录名单。有些数据库系统使用一个主数据表来记录该系统里都容纳着哪些数据库，但MySQL里不存在这种结构。MySQL数据目录结构的简单性使得查看数据库清单就是列出MySQL数据目录中的内容，增加一个数据表来存放它只会不必要地增加开销。

CREATE DATABASE db\_name命令将在MySQL数据目录里创建一个名为db\_name的空目录。在UNIX系统上，新创建的目录的属主就是启动MySQL服务器时使用的登录账户，并且只能通过该登录账户去访问它。也就是说，CREATE DATABASE命令相当于在使用那个账户登录到MySQL服务器主机上之后执行下列shell命令：

```
% cd DATADIR
% mkdir db_name
% chmod u=rwx,go-rwx db_name
```

用一个空目录来代表一个新数据库可以说是最简单的做法了。与此形成鲜明对照的是，在其他的数据库系统里，即使是创建一个“空的”数据库也需要创建好几个控制文件或系统文件。

DROP DATABASE语句实现起来也同样容易。执行DROP DATABASE *db\_name*命令其实是从MySQL数据目录里把目录*db\_name*及包含在其中的数据表文件全都删除掉——这与下面这几条UNIX命令的执行效果几乎完全相同：

```
% cd DATADIR
% rm -rf db_name
```

也与下面这几条Windows命令效果几乎相同：

```
C:\> cd DATADIR
C:\> del /s db_name
```

DROP DATABASE语句与shell命令之间的区别如下：

- 如果使用的是DROP DATABASE命令，MySQL服务器仅通过查看有关文件的扩展名删除那些与数据表有关的文件。如果还在数据库目录里创建过其他文件，MySQL服务器将保留这些文件，而且目录本身也不会被删除。（这意味着该数据库的名字仍将继续出现在SHOW DATABASES命令给出的数据库名单里。）
- InnoDB数据表及其索引的内容都存放在InnoDB表空间里，而不是被存放为MySQL数据目录中的文件。如果某个数据库里包含着InnoDB数据表，就必须使用DROP DATABASE语句才能让InnoDB处理程序从InnoDB表空间里把该数据表删掉，用rm或del命令删除数据库目录的方法对InnoDB数据表是无效的。

### 10.2.3 MySQL数据表在文件系统里如何表示

MySQL支持以下几种针对不同数据表类型的处理程序：ISAM、MyISAM、MERGE、BDB、InnoDB和HEAP。MySQL中的每一个数据表在磁盘上至少被表示为一个文件，即存放着该数据表的结构定义的.frm文件；大部分数据表类型还有其他几个用来存放数据行和索引信息的文件。这些文件会随着数据表类型的不同而变化。（这里的讨论重点是不同的数据表类型在磁盘上的存储特点，它们在功能和行为方面的差异请参见第3章。）

#### 1. ISAM数据表

MySQL中最原始的数据表类型就是ISAM类型。在MySQL里，每个ISAM数据表用包含着该数据表的数据库目录里的三个文件来代表。这些文件的基本名与数据表的名字相同，扩展名则分别表明了有关文件的具体用途。比如说，名为mytbl的ISAM数据表将被表示为以下三个文件：

- mytbl.frm——定义文件，存放着该数据表的格式（结构）定义。
- mytbl.ISD——ISAM数据文件，存放着该数据表中的各个数据行的内容。
- mytbl.ISM——ISAM索引文件，存放着该数据表的全部索引的索引信息。

#### 2. MyISAM数据表

MySQL 3.23版本引入了MyISAM数据表类型作为ISAM类型的后继者，而ISAM类型现在已

经过时了。类似于ISAM处理程序，MyISAM处理程序也要使用三个文件来代表一个数据表，这三个文件的扩展名分别是.frm（结构定义文件）、.MYD（数据文件）和.MYI（索引文件）。

### 3. MERGE数据表

MERGE数据表其实是一个逻辑结构。它代表着由一组结构完全相同的MyISAM数据表所构成的集合；有关的查询命令将把它当做一个大数据表来对待。在数据库目录里，每一个MERGE数据表将被表示为一个.frm文件和一个.MRG文件，.MRG文件其实就是一份构成MERGE数据表的各MyISAM数据表的名单（每个MyISAM数据表的名字占一行）。

这种表示方法意味着可以用以下办法来改变MERGE数据表的结构定义：先用FLUSH TABLES命令刷新数据表缓存区，然后直接编辑.MRG文件来改变原来的MyISAM数据表名单（但我不推荐这种做法）。

### 4. BDB数据表

BDB处理程序用两个文件来代表每个数据表，其一是用来存放数据表结构定义的.frm文件，其二是用来存放数据表的数据和索引信息的.db文件。

### 5. InnoDB数据表

前面几种数据表类型都是用多个文件来表示一个数据表的。InnoDB数据表与它们有所不同。与一个给定InnoDB数据表直接对应的文件只有一个，即数据表的.frm结构定义文件，这个文件存放在包含着数据表的数据库目录里。所有InnoDB数据表的数据和索引都被存放到同一个专用的表空间里统一管理。一般来说，这个表空间本身将被表示为MySQL数据目录里的一个或者多个大文件。构成表空间的这些大文件将形成一个在逻辑上连续不断的存储区域，表空间的总长度等于各组成文件的长度之和。

### 6. HEAP数据表

HEAP数据表是创建在内存中的数据表。因为MySQL服务器把HEAP数据表的数据和索引都存放在内存里而不是存放在硬盘上，所以除相应的.frm文件外，HEAP数据表在文件系统里根本没有相应的代表文件。

## 10.2.4 SQL语句如何映射为数据表文件操作

每一种数据表类型都要使用一个.frm文件来保存数据表的结构定义，所以SHOW TABLES db\_name命令的输出与列出数据库目录db\_name中所有.frm文件的文件基本名所得到的清单是一样的。有些数据库系统使用一个注册表来记录某数据库里的所有数据表，但MySQL没有这样做——因为这没有必要，MySQL数据目录的层次结构已经把“注册表”隐藏在其中了。

要想创建一个MySQL所支持的任意类型的数据表，需要发出一条CREATE TABLE语句来定义数据表的结构。无论哪一种数据表类型，MySQL服务器都将创建一个.frm文件来保存数据表的结构定义的内部编码。MySQL服务器还会根据给定数据表的具体类型创建出其他必要的文件来。比如说，它将为一个MyISAM数据表创建出一个.MYD数据文件和一个.MYI索引文件；为一个BDB数据表创建出一个.db数据/索引文件。对于InnoDB数据表，InnoDB处理程序将在InnoDB表空间里为数据表初始化一些数据和索引信息。在UNIX系统上，为新数据表而创建的各个文件的属主和存取模式将被设置为只允许用来运行MySQL服务器的账户进行访问。

当发出一条ALTER TABLE语句的时候, MySQL服务器将对有关数据表的.frm文件重新进行编码以反映出这条语句所表明结构性变化, 还要对有关数据文件和索引文件的内容进行相应的修改。CREATE INDEX和DROP INDEX语句也会引起类似的动作, 因为MySQL服务器在内部是把它们当做等效的ALTER TABLE语句来处理的。改变InnoDB数据表的结构会引起InnoDB处理程序对InnoDB表空间里的数据表的数据和索引做出相应的修改。

DROP TABLE语句是通过删除代表着数据表的各有关文件而实现的。丢弃一个InnoDB数据表将使数据表在InnoDB表空间里占用的空间被标注为“未使用”。

对于某些数据表类型, 可以通过在相应的数据库目录里删除与数据表有关的各个文件的办法来手动地删除这个数据表。比如说, 假设mydb是当前数据库, mytbl是一个ISAM、MyISAM、BDB或MERGE数据表, 那么DROP TABLE mytbl语句就大致等效于下面这两条UNIX命令:

```
% cd DATADIR
% rm -f mydb/mytbl.*
```

也大致等效于下面这两条Windows命令:

```
C:\> cd DATADIR
C:\> del mydb\mytbl.*
```

对于InnoDB或HEAP数据表, 因为它们的某些组成部分在文件系统里没有实体性的文件来代表, 所以针对这两种数据表类型的DROP TABLE语句没有等效的文件系统级命令。比如说, InnoDB数据表在文件系统里只有一个相应的.frm文件, 用文件系统级命令删除这个文件将使该数据表在InnoDB表空间的数据和索引成为“流离失所的孤儿”。

### 10.2.5 操作系统对数据库和数据表命名的限制

MySQL对数据库和数据表的命名有其自己的一套命名规则。这套规则的详细内容见第3章, 把其中的要点归纳如下:

- 名字可以由当前字符集中的字母和数字字符以及下划线(\_)和美元符号(\$)构成。
- 名字的最大长度是64个字符。
- 从MySQL 3.23.6版本开始, 其他字符也可以出现在名字里, 但必须把它们用反引号引起来(例如`odd@name`)。如果想把保留字用做数据列的名字, 那最好用反引号把它们引起来。

可是, 因为数据库和数据表的名字将被MySQL用做相应的目录和文件的基本文件名, 所以数据库和数据表的名字往往还要遵守MySQL服务器在其上运行的操作系统中的文件系统命名规则的限制:

- 只能用合法的文件名字来给数据库和数据表起名。因为每种数据表类型在文件系统里至少都会被表示为一个.frm文件, 所以各种数据表类型都要遵守这一规定。比如说, 按照MySQL的命名规则, 数据库和数据表的名字里允许出现“\$”字符, 可如果操作系统不允许文件名里出现该字符, 也就不能把它用在数据库目录或数据表名字里了。在实际工作中, 文件名中的“\$”字符对UNIX或Windows本身都不构成问题, 但它很可能会让你在shell里直接使用这种名字来进行数据库管理操作时遇到大麻烦。比如说, UNIX shell大都把“\$”



字符当做一个特殊字符来使用，如果给数据库起的名字里带有这个字符——比如\$mydb，那么，当在命令行上使用这个名字时，shell就会把它解释为变量引用，如下所示：

```
% ls $mydb
mydb: Undefined variable.
```

如果遇到这种情况，就必须对这个“\$”字符进行转义或者使用引号来抑制其特殊含义，如下所示：

```
% ls \ $mydb
% ls '$mydb'
```

注意，如果打算使用引号，请使用单引号。双引号是不能抑制shell把“\$mydb”形式的名字解释为变量引用的。

- 数据库或数据表的名字里不允许出现路径名分隔符，把它用引号引起来也不行。比如说，因为UNIX和Windows路径名的各组成部分是分别用“/”和“\”来分隔的，所以这两个字符都不能出现在数据库和数据表的名字里。在这两种平台上都不允许使用这两个字符的原因是为了便于把数据库和数据表从一个平台迁移到另一个平台去。（比如说，如果允许在Windows系统上的数据表名字里使用“/”字符，就不能把数据表迁移到UNIX系统上了。）
- 虽然MySQL所允许的数据库和数据表名字的最大长度是64个字符，但名字的实际长度还要受到操作系统所允许的文件名长度的限制。这一般不会成为问题，但某些早期的System V系列UNIX系统要求文件名的长度不得超过14个字符。对于这种情况，数据库名字的最大长度就将是14个字符；而数据表名字的最大长度将是10个字符，因为必须为数据表文件名留出4个字符来容纳最末尾的一个句点和最多3个字符的扩展名。
- 基本文件系统是否区分字母的大小写情况也会影响到对数据库和数据表的命名。如果文件系统区分字母的大小写情况（UNIX就是一个典型），mytbl和MYTBL这两个名字就将代表不同的数据表。如果文件系统不区分字母的大小写（比如Windows或Mac OS X下的HFS+文件系统），mytbl和MYTBL就将被视为同一个数据表。当在区分文件名中的字母大小写情况的服务器上开发一个有可能会在今后被迁移到不区分文件名中的字母大小写情况的服务器上去的数据库时，请一定要注意这一点。

处理字母大小写问题的办法之一是固定使用大写或者小写字母来命名数据库和数据表。办法之二是把MySQL服务器变量lower\_case\_table\_names设置为1，这将产生两个效果：

- 在创建有关的磁盘文件之前，MySQL服务器会把数据表的名字转换为小写字母。
- 当在查询命令中引用某个数据表时，MySQL服务器会在到磁盘上寻找有关文件之前把它们的名字转换为小写字母。

这两个效果结合起来就是不再把数据库和数据表的名字按区分字母大小写情况的做法来处理。但大家在使用服务器变量lower\_case\_table\_names时还应该注意这样两个问题：第一，在MySQL 4.0.2版本之前，这个办法只适用于数据表的名字，不适用于数据库的名字；第二，必须在开始创建数据库或数据表之前而不是之后激活这个变量。如果在创建了包含有大写字母的名字之后才想起要设置这个变量，是不会有预期效果的，因为包含着大写字母的名字已经被保存到磁盘上去了。为了避免今后发生问题，应该把在激活这个变量之前给数据库或数据表起的包



含有大写字母的名字全部改成都是小写字母的名字。

### 10.2.6 影响数据表最大尺寸的因素

在MySQL里,数据表的尺寸是有一定限制的。因为有好几个因素会影响到数据表的最大尺寸,所以有时很难精确地确定这个界限到底是多大。会影响到数据表最大尺寸的因素有以下几个:

- MySQL本身对数据表尺寸的限制,随数据表类型而变化:
  - 对于ISAM数据表,单个.ISD和.ISM文件的最大尺寸都是4GB。
  - 对于MyISAM数据表,单个.MYD和.MYI文件在默认情况下的最大尺寸也都是4GB。但可以在创建数据表的时候利用AVG\_ROW\_LENGTH和MAX\_ROWS选项把这个最大尺寸扩大到800万TB(请参见附录D中对CREATE TABLE语句的说明)。
  - MERGE数据表的最大尺寸是它的各组成MyISAM数据表的尺寸之和。
  - BDB数据表的尺寸受限于BDB处理程序所允许的.db文件的最大尺寸。这个最大尺寸随着数据表页面尺寸(这个尺寸是MySQL服务器在被编译建立时确定下来的)而变化。但即便是最小的页面尺寸(512字节),.db文件的最大尺寸也能达到2TB字节。
  - 对于InnoDB数据表,InnoDB表空间的尺寸是40亿个页面,默认的页面尺寸是16KB(在从源代码开始重新编译MySQL软件的时候,InnoDB页面尺寸可以被设置为8KB~64KB之间的某个值)。表空间的尺寸也就是任意一个InnoDB数据表的尺寸的上限。
- 操作系统对文件最大尺寸的限制。一般来讲,操作系统的发展趋势是放宽文件尺寸方面的限制,但把文件的最大尺寸限制为2GB的情况还是相当普遍的。这个尺寸限制同样适用于那些表示数据表的文件,比如用来表示MyISAM数据表的.MYD和.MYI文件等。它还适用于那些构成InnoDB表空间的组成文件。不过,InnoDB表空间的总长度很容易绕过单个文件最大尺寸方面的限制;只需把InnoDB表空间配置成由多个尺寸是单个文件最大尺寸的文件组成即可。绕过单个文件最大尺寸限制的另一种办法是使用未经格式化的硬盘分区来作为InnoDB表空间,在MySQL 3.23.41及以后的版本里都可以这样做。未经格式化的硬盘分区上的InnoDB表空间组成文件可以增长到与该分区本身的尺寸一样大。
- 对于那些要使用不同的文件来分别存放数据和索引的数据表类型(如ISAM和MYISAM),只要任何一个有关文件的尺寸达到单个文件的最大尺寸,数据表也就达到了它的最大尺寸。数据表的索引特性将决定哪个文件最先到达其尺寸的上限。比如说,如果某个数据表没有或只有很少几个索引,就可能是其数据文件最先到达其尺寸上限;但如果数据表有很多个索引,那其索引文件就可能先达到其尺寸上限。
- AUTO\_INCREMENT数据列对数据表中的数据行的个数也有隐含的限制。比如说,如果某个数据表里有一个TINYINT UNSIGNED类型的AUTO\_INCREMENT数据列,那么,因为该类型的最大可取值是255,所以这个数据表最多只能有255个数据行。更大的整数类型允许数据表有更多的数据行。一般而言,数据表里的惟一化索引将隐含地对数据表里的数据行最大个数做出限制,惟一化索引有多少种彼此不同的可取值,数据表的最大数据行个数

就将是多少。

要想确定某数据表的最大尺寸，必须把上述因素都考虑进来。数据表的实际最大尺寸通常要由上述因素中的最小值决定。以ISAM表为例，MySQL允许其数据文件和索引文件分别达到4GB，但如果操作系统规定单个文件的最大尺寸是2GB，那么2GB就将是数据表文件的实际最大尺寸。反之，如果操作系统支持单个文件的尺寸大于4GB，那么决定数据表最大尺寸的因素就将是MySQL的4GB内部限制。

至于InnoDB数据表，必须记住所有的InnoDB数据表都是容纳在同一个InnoDB表空间里的。如果只有一个InnoDB数据表，它就能增大到和表空间一样大。但如果有多多个InnoDB数据表（这种情况更常见），因为它们将共享同一个InnoDB表空间，所以每个InnoDB数据表不仅要受到InnoDB表空间尺寸的限制，还要看其他InnoDB数据表占用了多少InnoDB表空间。只要InnoDB表空间还有可用的空间，任何一个InnoDB数据表都可以增大。但是，当InnoDB表空间没有可用空间时，各个InnoDB数据表就不能再增长了——除非又通过给InnoDB表空间增加新组成文件的办法加大了它的尺寸。（从MySQL 3.23.50版本开始，可以把InnoDB表空间的最后一个组成文件设置为可自动扩展的，这样，只要它没有超过操作系统所规定的单个文件最大尺寸且磁盘上还有可用空间，InnoDB表空间就能不断增长。InnoDB表空间的配置办法详见第11章。）

### 10.2.7 数据目录的结构对系统性能的影响

MySQL数据目录的结构很容易理解，因为它以一种相当自然的方式使用了文件系统的层次结构。但这种结构对性能也有着隐含的影响。因为每打开一个数据表都需要打开一个或者多个与该数据表相关联的文件，所以这种层次结构对性能往往会产生很大的影响。

MySQL数据目录的结构后果之一是数据表处理程序需要使用多个文件来代表一个数据表，而打开一个数据表需要用到多个而不是一个文件描述符。MySQL服务器智能化地把文件描述符缓存起来，但一个繁忙的MySQL服务器仍有可能因为需要同时向多个客户连接提供服务或者执行一些涉及多个数据表的复杂查询而耗尽文件描述符。这很可能会是一个问题，因为文件描述符在许多系统——尤其是那些把每个进程所能使用的文件描述符个数（即允许每个进程打开的文件个数）设置得相当低的系统——上都是一稀缺的资源。

用多个文件来代表一个数据表的做法还会导致数据表打开操作的时间随查询命令所涉及的数据表个数的增加而延长。因为MySQL里的数据表打开操作将被映射为操作系统中的文件打开操作，所以系统的目录搜索例程的执行效率就会影响到数据表打开操作的执行效率。这一般算不上是个问题，但如果数据库里有很多个数据表的话，这个因素就很值得考虑了。比如说，我们知道，每个MyISAM数据表都将被表示为3个文件。假如数据库里有10 000个MyISAM数据表，则数据库目录就将有30 000个文件。要是真有这么多个文件，你就肯定会注意到文件打开操作明显地慢了下来（Linux ext2和Solaris文件系统会遇到这个问题）。如果系统性能因此而下降到足以引起关注的程度，就应该考虑换用一种能够高效率地处理大量文件的文件系统。比如说，ReiserFS文件系统就是一种在有大量小文件的情况下依然有着良好性能的文件系统。如果无法换

用另一种文件系统，就应该根据应用软件的具体情况重新考虑那些数据表的结构，对数据表的结构做出必要的优化调整。想想自己是否真的需要用到这么多数据表——有些应用软件会毫无必要地创建出很多数据表来。比如说，如果应用软件要为每个用户分别创建一个数据表，数据库里就会有结构完全相同的数据表。要想把这些数据表合并为一个数据表，可能需要增加一个数据列以表明各有关数据行都对应于哪一位用户。如果能大大减少数据表的数量，应用软件的性能就会有明显的改善。

在数据库系统的规划设计阶段，一定要考虑这种合并策略是否值得用在应用软件里。不把多个数据表合并为一个的理由主要有以下几点：

- **这会增加磁盘空间的占有量。**把多个数据表合并为一个数据表能减少数据表的个数（从而减少数据表打开操作的执行时间），但需要增加一个数据列（增加了磁盘空间占用量）。这是一个典型的“以空间换时间”的格局，必须确定哪个因素最重要。如果速度是重中之重，多付出一些磁盘空间做代价就很值得；可如果磁盘空间已经很紧张了，那么使用多个数据表并忍受一些延迟的做法大概更容易为你所接受。
- **安全方面的考虑。**安全方面的因素可能会限制你打算把多个数据表合并为一个数据表的能力或想法。为每个用户单独创建一个数据表的好处之一是可以利用一些数据表级的访问权限控制措施来保证每个用户只能访问到他自己的数据表。要是把这些数据表合并为一个，所有用户的数据就都汇集在同一个数据表里了。

MySQL不具备数据行级的访问控制机制，只要有权访问某个数据表，任何一位用户都能看到数据表里的全部信息；因此，想在不放弃访问控制的前提下把多个数据表合并为一个就不太容易做到。不过，如果对数据的访问是由应用软件来控制的（即用户不可能直接连接到数据库时），就能把多个数据表合并为一个并利用应用软件中的逻辑对合并结果进行数据行级的访问控制。

用尽可能少的文件创建出多个数据表的办法是使用InnoDB数据表。每个InnoDB数据表只有一个与之相关联的.frm文件，所有InnoDB数据表的数据和索引信息都将存放在InnoDB表空间里。这就大大减少了用来表示InnoDB数据表的磁盘文件的数量，进而减少了打开一个数据表时需要用到的文件描述符的个数。当用到InnoDB数据表的时候，InnoDB处理程序会先打开构成InnoDB表空间的各个组成文件（InnoDB表空间由几个组成文件构成，就要用到几个文件描述符；但这个数字在服务器运转期间将是一个常数）；然后，每打开一个InnoDB数据表，InnoDB处理程序就要用一个文件描述符去打开该数据表的.frm文件，每关闭一个InnoDB数据表，与该数据表相关联的文件描述符就会被释放。

#### 10.2.8 MySQL状态文件和日志文件

除数据库目录外，MySQL数据目录里还包含着许多状态文件和日志文件，如表10-1所示。这些文件的默认存放位置是相应的MySQL服务器的数据目录，其默认文件名是在服务器主机名（即表中的HOSTNAME）上增加一些后缀而得到的。

表10-1 MySQL的状态文件和日志文件

文件类型	默认名	文件内容
进程ID文件	<i>HOSTNAME.pid</i>	MySQL服务器进程的ID
常规查询日志	<i>HOSTNAME.log</i>	连接/断开连接事件和查询信息
慢查询日志	<i>HOSTNAME-slow.log</i>	耗时很长的查询命令的文本
变更日志	<i>HOSTNAME.nnn</i>	创建/变更了数据表的结构定义或者修改了数据表内容的查询命令的文本
二进制变更日志	<i>HOSTNAME-bin.nnn</i>	创建/变更了数据表的结构定义或者修改了数据表内容的查询命令的二进制表示法
二进制变更日志的索引文件	<i>HOSTNAME-bin.index</i>	使用中的“二进制变更日志文件”的清单
错误日志	<i>HOSTNAME.err</i>	“启动/关机”事件和异常情况

### 1. 进程ID文件

MySQL服务器会在启动时把自己的进程ID (PID) 写入PID文件, 等结束运行时又会删除该文件。这个PID文件的作用是向其他进程提供一种能够查知MySQL服务器进程ID的手段。比如说, 在系统关机过程中, 操作系统需要运行mysql.server脚本以停止MySQL服务器的运转, 这个脚本将查看MySQL服务器的PID文件以确定需要哪个进程发送一个“结束运行”信号。

### 2. MySQL日志文件

MySQL能够维护多个不同的日志文件。大多数日志功能都是可选的; 不仅可以在启动MySQL服务器时利用各种启动选项来只激活想启用的日志 (用不着的日志可以不启用), 还可以指定日志文件的名称 (如果你不喜欢它们的默认名称的话)。本小节只是对日志文件做一个简单介绍, 关于日志文件以及MySQL服务器用来控制日志行为的各种选项的详细介绍请参见第11章。

常规日志记录着服务器操作的综合性信息: 哪些人正从哪些地方试图连接MySQL服务器、他们发出了哪些查询命令等等。变更日志也记录着查询命令信息, 但它只记录那些对数据库内容做出了修改的查询命令。变更日志的内容是一些SQL语句, 可以把这些语句提供给mysql客户程序作为输入以执行之。二进制变更日志与变更日志作用相同, 但其内容是用效率更高的二进制格式写出来的。附属的二进制日志索引文件列出了MySQL服务器当前正维护着哪些二进制日志文件。

变更日志和二进制变更日志主要用在MySQL数据库系统的崩溃恢复工作中: 在发生崩溃后, 先用备份文件把数据库恢复到当初进行备份时的状态, 再把变更日志或二进制变更日志的内容馈入MySQL服务器, 让它再次执行日志中记载的各种修改操作, 把数据库恢复到崩溃发生时所处的状态。崩溃恢复工作的具体流程将在第13章中详细介绍。在建立镜像服务器的时候也要用到二进制变更日志, 因为这个日志的内容正是需要从主服务器传输到从服务器去的那份修改记录。

下面是常规日志中的一段信息示例, 这段信息表明: 有位用户连接到test数据库, 创建了一个数据表, 在其中插入了一个数据行, 最后丢弃了这个数据表并断开了连接:

```
020727 15:00:17      1 Connect      sampadm@localhost on test
                    2 Query        CREATE TABLE mytbl (val INT)
                    2 Query        INSERT INTO mytbl VALUES(1)
                    2 Query        DROP TABLE mytbl
                    2 Quit
```



常规日志中的信息由以下几项内容组成：日期/时间、服务器线程（连接）ID、事件的类型、事件的具体信息。如果后一事件与前一事件的发生日期/时间相同，则后一事件在常规日志中的记录项将省略日期/时间字段。（也就是说，只有当后一事件与前一事件的发生日期/时间不一样时，MySQL服务器才会在常规日志里记下一个新的日期/时间。）

下面是刚才那些操作在变更日志中的记载内容。请注意各SQL语句末尾的分号，这些分号使你能够——当需要在数据恢复工作中再次执行有关的变更操作（修改某数据表的结构或者修改某数据表的内容）时——把它们直接用做mysql客户程序的输入。

```
use test;
CREATE TABLE mytbl (val INT);
INSERT INTO mytbl VALUES(1);
DROP TABLE mytbl;
```

对于变更日志，可以用--log-long-format选项让它以扩展格式来记录有关事件；扩展格式将增加两项记载内容：谁发出的查询、什么时间发出的查询。变更日志的扩展格式当然要消耗更多的磁盘空间，但好处是不必对照常规日志中的连接事件就能知道谁正在做什么。

下面是刚才那些操作在使用了扩展格式的变更日志中的记载内容：

```
# Time: 020727 15:00:17
# User@Host: sampadm[sampadm] @ localhost []
use test;
CREATE TABLE mytbl (val INT);
# User@Host: sampadm[sampadm] @ localhost []
INSERT INTO mytbl VALUES(1);
# User@Host: sampadm[sampadm] @ localhost []
DROP TABLE mytbl;
```

请注意，新增加的记载内容都写在了以“#”字符开头的行上。这样，当把变更日志馈入mysql客户程序以传递给MySQL服务器去执行时，它们将被解释为注释。

错误日志记载着MySQL服务器在发生异常情况时生成的诊断信息。如果MySQL服务器启动失败或意外退出，通常可以从这个日志里了解到其原因。

日志文件的尺寸有可能变得非常大，一定要保证它们不至于填满文件系统。可以定期使一些日志文件失效以保证它们使用的空间总量不会超过一定的界限。日志文件的维护工作请参见第11章。

因为日志文件中记载的查询命令里可能会有口令之类的敏感信息，所以应该注意加强日志文件的安全保护工作，避免它们遭受意外破坏或者被无关用户读取。比如说，下列日志项里就有root用户的口令，这一定不是想让任何人看到的信息：

```
020727 15:47:24      4 Query      UPDATE user SET
                        Password=PASSWORD('secret')
                        WHERE user='root'
```

在默认的情况下，日志将被写到MySQL数据目录里，所以加强日志文件的安全保护工作的主要措施之一就是不让非MySQL管理员的登录账户进入并看到MySQL数据目录里的内容。这方面的具体做法请参见第12章。



## 10.3 重新安置数据目录的内容

本章前面讨论了默认配置情况下的MySQL数据目录结构，即把所有的数据库、状态文件和日志文件都存放在MySQL数据目录下时的情况。不过，你很可能想把MySQL数据目录里的某些内容放到其他地方去。MySQL允许重新安置数据目录本身或者其中某些特定的元素。下面是这样做的几个理由：

- 可以把数据目录放到一个容量比默认的文件系统更大的文件系统上去。
- 如果数据目录所在的磁盘非常繁忙，可以把它放到一个不那么繁忙的驱动器上以便在物理设备间平衡磁盘活动量。同样，可以把数据库和日志文件安排到不同的驱动器上或者把数据库分布到不同的驱动器上。类似地，虽然InnoDB表空间在概念上是一个连续的大存储块，但完全可以把它的各个组成文件放到不同的驱动器上以改善其性能。
- 把数据库和日志文件安排到不同磁盘上去的做法有助于降低因一块磁盘发生故障而导致重大损失的可能性。
- 你想运行多个MySQL服务器并让它们各有各的数据目录。这也是绕开每个进程所能使用的文件描述符个数限制的办法之一，尤其是在无法通过重新配置系统内核的办法来加大系统允许每个进程使用的文件描述符的最大个数的时候。
- 有些系统把PID文件保存在一个专用的目录（比如/var/run）里。为保持系统操作的一致性，应该把MySQL的PID文件也放到指定地点去。类似地，如果系统使用/var/log目录作为日志文件的统一存放地点，就应该把MySQL的日志文件也放到这个目录里去。（注意：很多系统只允许root用户对专门用来存放日志文件的目录进行写操作。这意味着必须以root用户身份来运行MySQL服务器；从安全角度考虑，这并不是一个好主意。）

本节的其余部分将讨论MySQL数据目录的哪些部分可以重新安置以及如何对它们重新安置。

### 10.3.1 重新安置方法

重新安置MySQL数据目录或其中元素的方法有以下两种：

- 可以在启动MySQL服务器时指定一个选项，既可以在命令行上，也可以在某个选项文件中。比如说，如果想指定MySQL数据目录的位置，那么既可以在命令行上使用`--datadir=dir_name`选项来启动服务器，也可以把下列各行放入某个选项文件里：

```
[mysqld]
datadir=dir_name
```

一般来讲，对应于服务器选项的选项文件组的名字是[mysqld]，如例中所示。然而，根据具体情况，给这个选项文件组另起一个名字可能更合适。比如说，如果使用的是嵌入式MySQL服务器，给选项文件组起名为[embedded]就更好。或者，如果用mysqld\_multi脚本启动了多个MySQL服务器，那最好给各选项文件组依次起名为[mysqldn]，其中n是代表某特定MySQL服务器的一个整数。将在第11章介绍MySQL服务器的不同启动方法都适合用哪些选项组来配置，第11章还提供了一些关于如何运行多个MySQL服务器的指导意见。

- 可以先对打算重新安置的东西进行移动，然后在原位置创建一个指向新位置的符号链接。

这两种方法各有其适用性和局限性。表10-2列出了MySQL数据目录中允许重新安置的各个项目以及可用的重新安置方法。如果打算使用选项文件来完成MySQL数据目录的重新安置工作，可以把有关选项放在全局选项文件（即UNIX系统下的/etc/my.cnf文件、Windows系统下的C:\my.cnf文件或Windows系统目录中的my.ini文件）里。

表10-2 MySQL数据目录的重新安置方法

将被重新安置的实体	适用的重新安置方法
整个数据目录	启动选项或符号链接
某数据库目录	符号链接
某数据表	符号链接
InnoDB表空间文件	启动选项
PID文件	启动选项
日志文件	启动选项

还可以使用MySQL默认数据目录（即编译在MySQL服务器程序里的数据目录）中的选项文件my.cnf来完成MySQL数据目录的重新安置工作。如果运行有多个MySQL服务器，把它们各自专用的配置选项分别存放到它们各自的my.cnf选项文件里是一个很不错的建议。不过，因为某给定MySQL服务器只知道到它的默认数据目录里去寻找自己的my.cnf文件，所以如果曾重新安置过该MySQL服务器的数据目录的话，它就找不到自己的my.cnf文件了。（解决这一问题的办法之一是在移动数据目录后在原位置创建一个指向新位置的符号链接。）

### 10.3.2 评估重新安置的效果

在重新安置MySQL数据目录之前，应该先对其能否达到预期的效果做一下评估。建议用du、df和ls -l命令来了解磁盘空间的使用情况，但这一切必须以你对文件系统的布局结构有着深入的了解为前提。

下面这个示例揭示了MySQL数据目录重新安置工作中应该注意的一个常见陷阱。假定MySQL数据目录是/usr/local/mysql/data，因为df命令表明/var文件系统有着更多的空间（如下例所示），所以打算把这个数据目录移动到/var/mysql去：

```
% df /usr /var
Filesystem 1K-blocks    Used    Avail Capacity  Mounted on
/dev/wd0s3e   396895    292126    73018     80%    /usr
/dev/wd0s3f  1189359   1111924   162287     15%    /var
```

重新安置这个数据目录会在/usr文件系统上释放出多少空间呢？为了找出答案，我们先用du -s查看一下这个目录占用了多少空间：

```
% cd /usr/local/mysql/data
% du -s
133426 .
```

差不多有130MB，这将使/usr的可用空间发生重大变化。但实际情况又如何呢？再用df命令

来看一下这个数据目录的空间使用情况：

```
% df /usr/local/mysql/data
Filesystem    1K-blocks    Used    Avail Capacity  Mounted on
/dev/wd0s3f   1189359    1111924    162287     15%      /var
```

很奇怪，我们想查看的是MySQL数据目录所在的文件系统（即/usr）的空间使用情况，可df报告出来的却是/var文件系统上的空间使用情况。为什么会这样？下面这条ls -l命令解答了我们的疑惑：

```
% ls -l /usr/local/mysql/data
...
lrwxrwxr-x  1 mysqladm  mysqlgrp  10 Dec 11 23:46 data -> /var/mysql
...
```

这个输出表明，/usr/local/mysql/data已经是一个指向/var/mysql的符号链接了。换句话说，这个数据目录已经被重新安置到/var文件系统上并被替换为一个指向新位置的符号链接了——把这个数据目录移动到/var文件系统并不会使/usr文件系统增加多少可用空间！

这就说明，事先花上几分钟的时间去评估一下重新安置操作的效果是非常值得的。这耽误不了多长时间，却能让你避免在花费大量时间去重新安置什么东西之后才发现它根本达不到预期效果的尴尬局面。

#### 重新安置MySQL数据目录注意事项

在开始重新安置MySQL数据目录之前，应该先结束MySQL服务器的运行，等完成安置工作之后再重新启动它。严格地讲，有些东西（比如某个数据库目录）是可以在MySQL仍在运行时重新安置的，但这种做法不值得推荐。如果你执意这样做，请务必保证MySQL服务器不会在你正在移动某个数据库的同时去访问它。此外，在移动某个数据库之前，要先发出一条FLUSH TABLES语句以确保MySQL服务器把所有已打开的文件全都关闭了。如果不注意这些细节，就可能导致有关的数据表遭到破坏。

### 10.3.3 重新安置整个数据目录

在重新安置某MySQL数据目录的时候，先要关停相应的MySQL服务器，再把它的数据目录移动到新位置。接下来，既可以删除原来的数据目录并把它替换为一个指向新位置的符号链接，也可以用一个明确地给出新位置的--datadir选项去重新启动这个MySQL服务器。推荐使用在原位置创建一个符号链接的做法，因为如果那个数据目录里有一个my.cnf文件，相应的MySQL服务器还能找到它。

### 10.3.4 重新安置一个数据库

因为MySQL服务器只会在其数据目录里寻找数据库目录，所以只能采用创建一个符号链接的办法重新安置某个数据库。在UNIX系统上，请按以下步骤进行：

- 1) 关停正在运行的MySQL服务器。
- 2) 把数据库目录拷贝或移动到新位置。
- 3) 删除原来的数据库目录。
- 4) 在MySQL数据目录里用这个数据库原来的名字创建一个符号链接并让它指向新位置。
- 5) 重新启动MySQL服务器。

下面是按以上步骤把一个名为bigdb的数据库移动到另一个位置的例子：

```
% mysqladmin -p -u root shutdown
Enter password: *****
% cd DATADIR
% tar cf - bigdb | (cd /var/db; tar xf -)
% mv bigdb bigdb.orig
% ln -s /var/db/bigdb .
% mysqld_safe &
```

这些命令需要以MySQL管理员的身份登录才能执行。为防止意外，这里增加了一个把数据库bigdb改名为bigdb.orig的步骤。等证实MySQL服务器能够正确地找到并使用重新安置后的数据库之后，再用下面这条命令把原来的数据库删掉：

```
% rm -rf bigdb.orig
```

在Windows系统上重新安置某个数据库的步骤与上面的稍有不同：

- 1) 关停正在运行的MySQL服务器。
- 2) 把数据库目录移动到新位置。

3) 在MySQL数据目录里创建一个.sym文件并让它指向这个数据库的新位置。比如说，如果把数据库sampdb从C:\mysql\data\sampdb移到E:\mysql-book\sampdb，就要在C:\mysql\data里创建一个内容如下所示的sampdb.sym文件：

```
E:\mysql-book\sampdb\
```

这个.sym文件相当于一个符号链接，它将告诉MySQL服务器从何处找到被移动了的数据库目录。

4) 重新启动MySQL服务器，但别忘了激活符号链接支持功能。激活符号链接支持功能的办法有两个，一是在命令行上使用--use-symbolic-links选项；二是把以下语句添加到某个选项文件里去：

```
[mysqld]
use-symbolic-links
```

为保证按上述步骤重新安置的数据库能够在Windows下正确使用，所使用的MySQL软件必须是3.23.16或更高的版本且必须是-max服务器（mysqld-max或mysqld-max-nt）。

如果把数据库移动到另一个文件系统的目的是为了平衡磁盘的读写活动，请不要忘记InnoDB数据表的内容是存储在InnoDB表空间里而不是存储在数据库目录里的。如果某个数据库主要是由一些InnoDB数据表构成的，重新安置这个数据库目录可能不会让你在平衡磁盘活动方面有太大的收获。

### 删除重新安置的数据库

可以用DROP DATABASE语句来删除一个数据库，但3.23版本之前的MySQL服务器在删除一个已重新安置的数据库时会遇到一些麻烦——数据库里的数据表都能被正确地删除，但服务器在试图直接删除数据库目录时会出现错误。这是因为那个目录只是一个符号链接而不是一个真正的目录。在遇到这类问题的时候，必须以手动方式删掉那个数据库目录和指向它的符号链接才能真正完成DROP DATABASE操作。

### 10.3.5 重新安置一个数据表

只有在以下条件全部满足时才能重新安置一个数据表：

- 所使用的MySQL软件必须是4.0或更高的版本。
- 操作系统必须具有一个确实能工作的realpath()调用。
- 打算重新安置的数据表必须是一个MyISAM数据表。

只有上述条件全部得到满足时，才能把某个MyISAM数据表的.MYD数据文件和.MYI索引文件移动到新位置去——别忘了在原来的数据文件和索引文件所在的数据库目录里创建两个符号链接分别指向它们的新位置。（MyISAM数据表的.frm定义文件仍需留在原来的数据库目录里。）

若上述条件不能全部满足，最好不要重新安置数据表。如果你一意孤行地这样做了，那么，今后只要你一使用ALTER TABLE、OPTIMIZE TABLE或REPAIR TABLE语句对重新安置后的数据表进行修改或优化，它就会回到原来的位置，使你对它的重新安置失去效果。导致这一问题的原因是ALTER TABLE、OPTIMIZE TABLE或REPAIR TABLE语句的具体执行过程：先在MySQL数据目录里创建一个临时数据表并对这个临时数据表进行修改或优化；然后删掉原数据表，再把临时数据表更名为原数据表的名字。这样，如果你对一个重新安置过的数据表使用了ALTER TABLE、OPTIMIZE TABLE或REPAIR TABLE语句，就会形成这样的局面：当初在重新安置数据表时创建的符号链接将被删除，而由临时数据表更名得到的新数据表将重新出现在老数据表未被移动之前所在的数据库目录里；与此同时，从数据库目录里移出来的老数据表文件却仍在当初安置它们的位置上——没人会提醒你说它们仍在那里占用着磁盘空间。因为当初创建的符号链接已经被删掉了，所以，等你意识到所发生的事情时，要是你自己也没记住当初把老数据表安置到了什么地方，就很难再追踪到那些老数据表文件了。在一个多用户的系统上，即使你本人在重新安置过某个数据表后没有对它使用过ALTER TABLE、OPTIMIZE TABLE或REPAIR TABLE语句，也很难保证有权访问这个数据表的其他用户不会这样做（并因此而导致它回到了原来的位置），所以，如无特殊的必要，还是把数据表留在其原来所在的数据库目录里比较稳妥。

### 10.3.6 重新安置InnoDB表空间

InnoDB表空间是通过在某个选项文件里使用innodb\_data\_home\_dir和innodb\_data\_file\_path



选项列出一份InnoDB表空间组成文件清单的办法而配置出来的（InnoDB表空间的详细配置步骤见第11章）。在创建出InnoDB表空间之后，你可能会遇到需要重新安置它的某个组成文件的情况，比如说，如果你想平衡磁盘的读写活动，就可能需要把InnoDB表空间的组成文件均匀地分散到不同的文件系统去。既然这些组成文件的位置是通过MySQL服务器的启动选项配置出来的，仍通过启动选项来重新安置它们就是一件顺理成章的事情了：

- 1) 关停正在运行的MySQL服务器。
- 2) 移动打算重新安置的一个或者多个表空间组成文件。
- 3) 修改当初用来配置InnoDB表空间的选项文件，把组成文件的新位置列成一份新的清单。
- 4) 重新启动MySQL服务器。

严格地讲，采用“先移动有关文件，再在它们的原位置创建一个指向新位置的符号链接”的办法来重新安置InnoDB表空间的某个组成文件也不是不可以，但这种做法有点画蛇添足了：反正要在选项文件里写出各组成文件的位置，干脆直接写出它们的新位置，没必要使用符号链接。

### 10.3.7 重新安置状态文件和日志文件

重新安置PID文件的步骤是：先关停MySQL服务器，再用一个能够设定PID文件新位置的选项重新启动它。比如说，如果想把PID文件设定为/tmp/mysql.pid文件，有两个办法，其一是在命令行上使用--pid-file=/tmp/mysql.pid选项；其二是在某个选项文件里添加以下内容：

```
[mysqld]
pid-file=/tmp/mysql.pid
```

如果PID文件名是以绝对路径名的形式给出的，MySQL服务器就将使用该路径名去建立PID文件；否则，MySQL服务器就将把PID文件创建在它自己的数据目录中。比如说，如果给出的是--pid-file=mysql.pid，PID文件就将是相应的MySQL数据目录中的mysql.pid文件。

日志文件可以利用MySQL服务器的启动选项来重新安置，我们将在第11章对有关选项做详细介绍。



## 第11章 MySQL数据库系统的日常管理

本章讨论作为MySQL管理员如何把MySQL运行得更平稳。MySQL管理员的基本职责包括：顺利启动MySQL服务器并让它尽可能长时间地持续运转；创建用户账户，使其他人能够访问MySQL服务器；对日志文件进行维护。此外，还需要知道如何修改MySQL服务器的操作参数才能获得更好的性能、才能运行多个服务器或者在MySQL服务器之间建立镜像。最后，因为MySQL本身仍在不断地开发和完善当中，所以MySQL管理员必须知道何时需要安装新版本以升级MySQL软件本身。还有一些重要的MySQL管理员职责将在第12章和第13章中介绍。

本章将对MySQL管理员必须熟练运用的几个管理工具程序进行介绍，它们是：

- `mysqld`：MySQL服务器程序。
- `mysqld_safe`、`mysql.server`和`mysqld_multi`：MySQL服务器启动脚本（`mysqld_safe`脚本在MySQL 4之前的版本里叫做`safe_mysqld`）。
- `mysqladmin`：用来完成各种管理性操作的工具程序。
- `mysqldump`和`mysqlhotcopy`：数据库备份和拷贝工具。
- `mysqlcheck`、`myisamchk`和`isamchk`：用来完成数据表完整性检查和修复操作的工具。

本章大部分内容都需要读者对MySQL服务器用来存放各种数据库、日志文件以及其他信息的MySQL数据目录有着较深的理解才能更好地领会和运用。对MySQL数据目录的讨论详见第10章。附录D和附录E对本章涉及的SQL语句和程序做了详细的说明。

### 11.1 新MySQL软件的安全措施

MySQL软件的安装过程将创建一个MySQL数据目录，并在其中建立两个数据库：

- `mysql`数据库，它包含着各种权限表。
- `test`数据库，供测试目的使用。

在刚安装好MySQL软件（具体步骤见附录A）的时候，`mysql`数据库中的权限表都处于初始状态，允许任何人不用提供口令就能连接到MySQL服务器。这是不安全的，所以所要做的第一件事就是设置一些口令。如果你的机器里已经安装有一套MySQL软件、现在在这台机器上安装的是第二套MySQL软件，将需要为新安装的MySQL服务器设置一些口令。不过，在后一种情况下，你可能会遇到第11.1.3节中提到的困境。如果你是在对MySQL软件进行升级，即用一套新版本的MySQL软件来替换你现在使用的MySQL软件，因为现有的权限表可以继续沿用，所以可以跳过这一节。

在本章给出的用法示例中，我们将以`cobra.snake.net`作为MySQL服务器的主机名；大家在按有关步骤进行操作时请把它改为你自己的主机名。因为需要先连接上MySQL服务器才能进行管理性操作，所以本章中的示例都假设你的MySQL服务器已经启动并正在运行。

### 11.1.1 权限表的初始设置情况是怎样的

mysql数据库里的权限表是在MySQL软件的安装过程中用以下两类账户创建出来的：

- 一些以root为用户名的账户。它们是用来完成各种数据库管理操作的超级用户账户。它们拥有进行各种操作（包括删除全部的数据库）的权限。（顺便说一句，MySQL和UNIX的超级用户账户都以root为名纯粹是巧合。虽然都拥有至高的权限，但它们之间毫不相干。）
- 一些根本没有用户名的账户，即所谓的“匿名”账户。匿名账户主要用于测试目的，任何人都可以使用匿名账户去连接MySQL服务器而无需由MySQL管理员事先为他们明确地创建一个账户。一般说来，避免因试用者操作不当而对系统造成破坏，匿名账户通常只有很少的权限。但Windows系统上的匿名MySQL用户却有着相当高的权限级别，所以Windows系统上的MySQL管理员必须采取一些措施来改善匿名MySQL用户的安全性，这个问题稍后还将做详细介绍。

MySQL服务器把它知道的每一个账户都列在mysql数据库里的权限表user里，所以到那儿可以找到MySQL安装过程所创建的root用户和匿名账户。由MySQL安装过程创建的这些初始账户全都没有口令，MySQL把设置口令的事留给了MySQL管理员去完成。因此，在安装好MySQL软件之后，所要做的第一件事应该是为初始账户——至少是那些特权账户——设置口令以确保低权限用户不会轻易获得root权限；否则，就等于是让自己的系统暴露在有意或者无意的破坏和攻击面前。在给初始账户设置好口令之后，可以再创建一些其他的账户以便用户能够使用你指定的用户名去连接MySQL服务器并在你划定的权限范围内去完成自己的日常工作。（设置新账户的具体步骤见第11.3节。）

user数据表里的各条记录都包含Host（主机名，使用该账户的用户可以从什么地方连接MySQL服务器）、User（使用该账户的用户在连接服务器主机时给出的用户名）、Password（使用该账户的用户在连接服务器主机时给出的口令）等几项数据。user数据表还有一些用来表明有关账户具备何种超级用户权限（如果这个账户有超级用户权限的话）的数据列。

在UNIX系统上，MySQL数据目录的初始化工作是由mysql\_install\_db脚本在MySQL软件安装过程中完成的，这个脚本还将对mysql数据库里的权限表进行初始化。mysql\_install\_db脚本将把user权限表初始化成如下所示的样子：

Host	User	Password	Superuser Privileges
localhost	root		All
cobra.snake.net	root		All
localhost			None
cobra.snake.net			None

这些记录项允许人们以下面几种方式去连接MySQL服务器：

- root记录项允许以localhost或cobra.snake.net作为主机名去连接本地主机上的MySQL服务器。比如说，在主机cobra.snake.net上，下面两条命令都能让你使用mysql程序以root身份连接上MySQL服务器：

```
% mysql -h localhost -u root
% mysql -h cobra.snake.net -u root
```

作为root用户，将拥有全部的权限并能执行任何一种操作。

- User部分是空白的记录项就是匿名账户。它们能让你在不必给出任何用户名的情况下连接到本地主机中的MySQL服务器上：

```
% mysql -h localhost
% mysql -h cobra.snake.net
```

从user权限表看，匿名用户不具备超级用户权限；但另一个权限表（db权限表，这里没有给出其内容）却允许匿名用户访问test数据库和名字以“test”开头的其他数据库。

在用来往Windows系统上安装MySQL软件的MySQL发行版本里，MySQL数据目录和mysql数据库都是事先预初始化好了的，账户设置情况与UNIX系统上的有所不同。下面是Windows系统上的user权限表的内容：

Host	User	Password	Superuser Privileges
localhost	root		All
%	root		All
localhost			All
%			None

在这些记录项里，Host一栏中的“%”相当于一个通配符，意思是由User值指定的用户可以从任何一台主机去连接MySQL服务器。也就是说，Windows系统上最初的user权限表设置了以下几种账户：

- 可以从本地主机或任何远程主机以root身份连接MySQL服务器。作为root用户，将拥有全部的权限并能执行任何一种操作。
- 不必给出任何用户名就能以匿名方式连接上MySQL服务器。如果是从本地主机去连接MySQL服务器的，就将拥有与root用户完全一样的超级用户权限。如果是从另一台主机远程连接到MySQL服务器的，将不具备超级用户权限。Windows系统上的db权限表允许匿名用户访问test数据库和名字以“test”开头的其他数据库。

请注意，在Windows系统上的初始root账户中，有一个Host值是“%”。这一事实意味着任何人从任何地方都能够不必给出口令就以root身份连接上MySQL服务器。这等于是把MySQL服务器完完全全地暴露在了危险当中，所以千万要记住给它设置一个口令以保护好那个账户。此外，Host值是“localhost”的匿名账户具有与root用户同样的权限，所以只给root账户设置口令是不够的。要想堵住这一安全漏洞，可以给这个本地匿名账户也设置一个口令、收回它的超级用户权限或者干脆把这个账户删除掉。下面的讨论包括全部三个选项。

### 11.1.2 为MySQL初始账户设置口令

本小节将向大家介绍几种为root账户设置口令的具体办法。根据所选用的具体办法，可能还需要告诉MySQL服务器去重新加载权限表才能让你做出的修改生效。（MySQL服务器是使用驻留在内存里的权限表拷贝来进行访问控制的。如果你直接去修改user权限表中的口令，MySQL服务器可能不知道你修改了些什么，所以必须明确地告诉它去重新读取权限表。）我们还将在这一小节里向大家介绍几种用来处理Windows系统上的user权限表里的初始匿名超级用户账户的方法。

设置口令的第一种办法是使用mysqladmin程序：

```
% mysqladmin -h localhost -u root password "rootpass"
% mysqladmin -h cobra.snake.net -u root password "rootpass"
```

这个办法适用于UNIX和Windows。这两条命令中的“password”表明想让mysqladmin程序去设置一个口令，“rootpass”代表着所设置的口令值。这两条mysqladmin命令要同时使用；第一条命令用来给Host值是localhost的root账户设置口令，第二条命令用来给Host值是cobra.snake.net的root账户设置口令。（在Windows系统上，第二条命令将给Host值是“%”的root账户设置口令。）

设置口令的第二种方法是发出SET PASSWORD语句。在SET PASSWORD语句里，某用户在user权限表中的用户名（User项）和主机名（Host项）应写成'user\_name'@'host\_name'格式。下面是用在UNIX系统上修改账户口令的有关命令：

```
% mysql -u root
mysql> SET PASSWORD FOR 'root'@'localhost' = PASSWORD('rootpass');
mysql> SET PASSWORD FOR 'root'@'cobra.snake.net' = PASSWORD('rootpass');
```

下面是用在Windows系统上修改账户口令的有关命令（请注意第二条SET语句里的主机名部分）：

```
C:\> mysql -u root
mysql> SET PASSWORD FOR 'root'@'localhost' = PASSWORD('rootpass');
mysql> SET PASSWORD FOR 'root'@'%' = PASSWORD('rootpass');
```

设置口令的第三种办法是直接修改user权限表。这个办法适用于MySQL的所有版本；事实上，如果你使用的MySQL软件是mysqladmin password命令和SET PASSWORD语句出现之前的某个老版本，则只有这个办法可用。下面的语句能同时完成两个root账户的口令设置工作：

```
% mysql -u root
mysql> USE mysql;
mysql> UPDATE user SET Password=PASSWORD('rootpass') WHERE User='root';
mysql> FLUSH PRIVILEGES;
```

如果使用mysqladmin password命令或SET PASSWORD语句来改变口令，MySQL将觉察到你对权限表进行了修改，它将自动读入权限表的新内容去刷新内存里的拷贝；如果使用UPDATE语句直接修改user权限表，就必须明确地通知MySQL服务器重新加载这个权限表。在MySQL 3.22.9及以后的版本里，可以用FLUSH PRIVILEGES语句来刷新权限表，就像上例那样。还可用mysqladmin程序来重新加载权限表：

```
% mysqladmin -u root reload
% mysqladmin -u root flush-privileges
```

reload适用于MySQL的任何版本；flush-privileges则是从MySQL 3.22.12版本才开始出现的。从现在起，当提到“重新加载权限表”的时候，表示使用上面三种办法之一，具体使用哪一个没有太大关系。（本章后半部分的操作示例将以使用FLUSH PRIVILEGES语句为主。）

在设置好root口令（并视情况重新加载了权限表）之后，再想以root身份去连接MySQL服



务器就必须给出新口令了：

```
% mysql -p -u root
Enter password: rootpass
mysql>
```

为root账户设置口令还有另一个作用，即不知道这个口令的人将无法以root用户的身份去连接MySQL服务器，而这一点才是我们要为root账户设置口令的真正目的。

从现在起，当你想以root身份连接到MySQL服务器去的时候，就必须给出该账户的口令；这不仅适用于mysql程序，也适用于mysqladmin、mysqldump等程序。为简洁起见，本章的很多示例都省略了-u和-p选项；但大家在以root用户身份连接MySQL服务器或者试用有关命令的时候千万不要忘记把口令加进去。

接下来，给user权限表里的匿名账户设置口令。如果根本不需要匿名账户，可考虑把它们全都删掉。具体做法是：以root用户身份连接到MySQL服务器（不要忘记给出新口令），把user和db权限表里User值是空白的记录项全都删掉，然后重新加载权限表，如下所示：

```
% mysql -p -u root
Enter password: rootpass
mysql> USE mysql;
mysql> DELETE FROM user WHERE User = '';
mysql> DELETE FROM db WHERE User = '';
mysql> FLUSH PRIVILEGES;
```

如果不想删掉匿名账户，别忘了Windows系统上的本地匿名账户与root用户是有着同样权限的，不应该在自己的数据库系统里留着一个这么大的安全漏洞。应该把本地匿名账户的权限削弱到与某个远程匿名账户差不多的程度：先以root用户身份连接到MySQL服务器，然后用下列语句撤销其超级用户权限：

```
mysql> REVOKE ALL ON *.* FROM ''@'localhost';
mysql> REVOKE GRANT OPTION ON *.* FROM ''@'localhost';
```

保护匿名账户的另一个办法是给它设置一个口令，如下所示：

```
mysql> SET PASSWORD FOR ''@'localhost' = PASSWORD('anonpass');
```

把匿名账户保留在user权限表里会导致一种奇怪的现象（将在第12.2.3节里对此做详细说明）。但大家先不要急于看那一节，等阅读了下面几小节内容、对如何设置口令有了更多了解之后再去看那一节不迟。

### 11.1.3 为第二个MySQL服务器设置口令

此前的讨论一直假设你是在一个以前从未安装过MySQL软件的系统上设置口令。现在，假设机器里已经安装有一套MySQL软件而你正在为新安装在同一台机器的另一个MySQL服务器设置口令。你可能会遇到这样的问题：当你在没有给出口令的情况下去连接新安装的MySQL服务器时，系统会拒绝你的连接请求并给出如下所示的出错信息：

```
% mysql -u root
ERROR 1045: Access denied for user: 'root@localhost' (Using password: YES)
```

奇怪！为什么根本没有给出口令，而服务器却会收到口令呢？导致这一问题的原因往往是这样的：在某个用来访问以前安装的MySQL服务器的选项文件里列出了一个口令；当你这次使用mysql程序去连接新安装的MySQL服务器时，mysql程序找到了那个选项文件并自动使用了那里列出的口令。要想绕开这一问题并明确地表明“这次连接用不着口令”，应该在命令行上多给出一个-p选项，然后在mysql程序提示输入口令的时候直接按下回车键，如下所示：

```
% mysql -p -u root
Enter password:          ← 直接按下回车键
```

这个办法也适用于mysqladmin以及其他MySQL客户程序。

关于在同一台机器上运行多个MySQL服务器的讨论见第11.6节。

## 11.2 安排MySQL服务器的启动和关闭

作为一名MySQL管理员，基本职责之一将是保证MySQL服务器能够尽可能长时间地连续运转，以便用户能够访问它。可有时必须把MySQL服务器暂时停下来。比如说，你肯定不希望MySQL服务器在你重新安置某个数据库的同时去修改该数据库里面的数据表。想让MySQL服务器尽可能长时间运行的愿望与把它暂时关闭的工作需要是矛盾的，而这个矛盾是本书无法解决的。这里只讨论如何启动和关闭MySQL服务器，这样，读者就能根据自己的实际工作情况来决定该如何去做了。MySQL服务器在UNIX和Windows系统上的启动和关闭步骤有很多不同之处，所以下面将把它们分开讨论。

### 11.2.1 在UNIX系统上运行MySQL服务器

在UNIX系统上，既可以在系统开机后手动启动MySQL服务器，也可以安排它在系统开机时自动启动。（事实上，在把该配置的东西都配置好了之后，让MySQL服务器在系统开机时自动启动可能是最常用的启动方式。）不过，在讨论如何启动MySQL服务器之前，我们先要说说应该使用哪个登录账户来启动和运行MySQL服务器。在UNIX这样的多用户操作系统上，可以选择使用某个登录账户来运行MySQL服务器。比如说，如果是以手动方式来启动MySQL服务器的，它就会运行在登录时所使用的UNIX用户名下。也就是说，如果登录为用户paul并启动MySQL服务器，它将运行在paul名下；如果先使用su命令切换为用户root再启动MySQL服务器，它就会运行在root名下。

在UNIX系统上启动MySQL服务器的时候，应该注意这样两件事：

- **最好别把MySQL服务器运行在root用户名下。**“某某服务器运行在某用户名下”的意思是该服务器的进程将与该用户的登录账户的用户ID关联在一起，因而具备该用户在文件系统里读、写文件的权限。这将牵涉到一些系统安全方面的问题，尤其是那些运行在root用户名下的进程——因为root用户允许做任何事情，所以让某个进程运行在root用户名下往往会对整个系统的安全构成威胁。（第12章将对运行在root用户名下的MySQL服务器可能导致的一些问题进行说明。）而避免这类危险的办法之一是让MySQL放弃它的某些特殊权限。在root用户名下启动的进程具备把自己的用户ID切换为另一个账户的能力，有关进程将放

弃root用户的某些特权而以那位非特权用户的权限运行。这就降低了有关进程对整个系统的安全性的威胁。一般说来,进程的权限越少越好;如果没有真正的必要,任何进程——特别是mysqld进程——都不应该运行在root名下。MySQL服务器需要访问和管理MySQL数据目录的内容,但也仅此而已。这意味着如果是以root用户身份去启动MySQL服务器的,就应该让它在启动过程中把自己切换为运行在某个非特权用户的名下。(但在Solaris系统上有一个例外:如果MySQL服务器被频繁交换出内存,可能需要使用--memlock选项来强制它驻留在内存里;该选项要求MySQL服务器必须运行在root用户名下。)

- 应该让MySQL服务器总是运行在同一个用户名下。让MySQL服务器一会儿运行在这个用户名下、一会儿又运行在另一个用户名下的做法是不好的,这将使MySQL服务器在其数据目录里创建出来的文件和目录分属不同的属主,甚至导致MySQL服务器无法访问某些数据库或数据表。让MySQL服务器一直运行在同一个用户名下可以避免这个问题。

#### 1. 使用非特权登录账户来运行MySQL服务器

用一个无特权的非root登录账户来启动和运行MySQL服务器的具体步骤如下:

##### 1) 如果MySQL服务器正在运行,先关停之:

```
# mysqladmin -p -u root shutdown
```

2) 选择一个可以用来运行mysqld的登录账户。可以使用任何账户,但为MySQL活动单独创建一个专用的账户在概念上和管理上将更清晰。还可以单独创建一个供MySQL专用的用户组名。在下面的讨论里,将以mysqladm和mysqlgrp作为供MySQL专用的用户名和用户组名。如果你使用了不同的名字,请在本书的有关示例中用它们替换mysqladm和mysqlgrp。比如说,如果你把MySQL安装在了自己的账户下而你在系统上又没有特殊的管理员权限,就只能把MySQL运行在你自己的用户ID下;对于这种情况,就要用你自己的登录名和用户组名替换掉本书示例中的mysqladm和mysqlgrp。如果你是用RPM文件把MySQL安装在Linux系统上的,那么安装过程就很可能已经自动创建一个供MySQL专用的账户了;对于这种情况,就要用那个账户的用户名和用户组名替换掉本书示例中的mysqladm和mysqlgrp。

3) 如有必要,请按照正常的账户创建步骤把所选择的登录账户创建出来。这项工作必须以root用户的身份来进行。

4) 修改MySQL数据目录及其下级子目录和文件的用户与用户组的属主项,把拥有它们的用户和用户组修改为mysqladm和mysqlgrp。比如说,如果MySQL数据目录是/usr/local/mysql/data,可以用下面这条命令来设置该目录及其内容的属主项(必须以root用户的身份来运行这个命令):

```
# chown -R mysqladm.mysqlgrp /usr/local/mysql/data
```

5) 出于安全方面的考虑,最好把MySQL数据目录的访问模式设置成不允许其他用户访问。这可以通过把MySQL数据目录的访问权限设置为只允许mysqladm用户使用它的办法来实现。比如说,如果MySQL数据目录是/usr/local/mysql/data,可以用下面这条命令把它的“group”和“other”访问权限全部去掉,让这个目录及其中的所有内容只允许mysqladm用户去访问:

```
# chmod -R go-rwx /usr/local/mysql/data
```

最后两个步骤其实是将在第12章中详细介绍的一套更全面的安全措施的组成部分。建议读

者——尤其是以定制方式安装MySQL软件的读者——现在就去看看第12章中的有关说明。

按上述步骤完成有关处理之后，今后再启动MySQL服务器时请千万不要忘记加上`--user=mysqladm`选项。这样，即使MySQL服务器是在root用户名下启动的，它也会把自己的用户ID切换为mysqladm。（这句提醒对以root身份手动地启动运行MySQL服务器和让MySQL服务器在系统开机过程中自动启动运行两种情况都是适用的。要知道，UNIX系统的开机操作是在UNIX的root用户名下进行的，在默认情况下，开机过程所启动的任何进程在运行中都将具备root权限。）确保MySQL服务器始终如一地在同一用户名下运行的办法是把这个选项放到一个选项文件里去。比如说，可以把下面两行语句放到/etc/my.cnf文件中：

```
[mysqld]
user=mysqladm
```

关于选项文件的详细介绍请参见第11.2.3节。

选项文件中的user选项设置语句是专为root用户准备的，所以当你登录为mysqladm用户并去启动MySQL服务器的时候，将看到一条表明“MySQL服务器无法切换它的用户ID并将运行在mysqladm用户名下”的警告信息。这正是你想做的事，所以用不着理会这个警告。

--user选项是从MySQL 3.22版本开始才增加到mysqld程序里的。如果你使用的是较早的版本，就得用su命令把运行在root名下的MySQL服务器切换到另一个账户名下。因为不同版本的su命令有着不同的调用语法，所以需要查一下你的系统手册。

## 2. 启动MySQL服务器的方法

在决定使用哪一个账户来运行MySQL服务器之后，在它的具体启动办法上还有几种选择。既可以手动地从命令行启动MySQL服务器，也可以安排它在系统开机时自动运行。下面是几种具体的做法：

- **直接运行mysqld程序。**这种做法极其少见，这里不打算对它做进一步的讨论。但如果你想知道MySQL服务器都支持哪些启动选项，`mysqld --help`将是一个很有用的命令。
- **使用mysqld\_safe脚本。**mysqld\_safe脚本会在启动MySQL服务器后继续监控其运行情况并在其死机时重新启动它。用mysqld\_safe脚本来启动MySQL服务器的做法在BSD风格的UNIX系统上很常见，非BSD风格的UNIX系统中的mysql.server脚本也会用到它。（mysqld\_safe脚本在早于MySQL 4的版本里叫做safe\_mysqld；本章对mysqld\_safe脚本的介绍和讨论同样适用于safe\_mysqld脚本。）

mysqld\_safe脚本将把由它启动的MySQL服务器输出的出错信息和其他诊断性信息重定向到相应的数据目录中的某个文件里去，这个文件就是第10章介绍的错误日志。mysqld\_safe会把错误日志文件的属主设置为--user选项所指定的用户，如果前后两次启动MySQL服务器时使用的--user值不一样，就可能会遇到这样的麻烦：mysqld\_safe脚本对错误日志的写操作执行失败并报告一条“permission denied”（权限不足）出错信息。这个问题的麻烦之处在于你在错误日志里根本找不出它的根源——因为它们根本就没有被写入错误日志。如果你怀疑自己遇到的是这个问题，请删除错误日志并重新执行mysqld\_safe脚本。

- **使用mysql.server脚本。**mysql.server脚本将调用mysqld\_safe脚本去启动MySQL服务器。mysql.server脚本有两个参数，一个是start，用来启动MySQL服务器；另一个是stop，用



来关停MySQL服务器。mysql.server脚本多见于System V风格的UNIX系统，其作用相当于mysqld\_safe脚本的一件“外衣”。（System V风格的UNIX系统在开机启动时经常要用到来自多个目录的脚本；这些目录各自对应于特定的运行级别[runlevel]，其内容则是系统进入或者退出运行级别时将要调用的各个脚本程序。）

- 使用mysqld\_multi脚本（同时运行多个MySQL服务器）。这种启动方式比其他办法都复杂，将在第11.6节里对此做详细讨论。

mysqld\_safe和mysqld\_multi脚本将被安装在MySQL安装路径下的bin目录里，它们在MySQL源代码发行版本里的存放位置是scripts目录。mysql.server脚本程序将被安装在MySQL安装路径下的share/mysql目录里，它在MySQL源代码发行版本里的位置是support-files目录。要想使用这几个脚本，就得把它们拷贝到适当的启动目录里并把它们设置为可执行。如果你机器里的MySQL是用一个从MySQL官方网站上下载得到的RPM文件安装的，安装过程将把mysql.server脚本以“mysql”为名拷贝到/etc/rc.d /init.d目录里去；如果你机器里的MySQL是用一个从RedHat公司的网站上下载的RPM文件安装的，安装过程将把一个类似的启动脚本安装为mysqld脚本。

把启动脚本安排到什么地方才能让它们在系统开机时自动执行要取决于所使用的操作系统。在研读下面给出的操作示例时，请注意区分它们是否适用于你的系统开机过程。

在BSD风格的系统上，用来在系统开机时启动各种服务的脚本文件大都存放在/etc目录里，这些文件的名字通常以“rc”开头。其中，应该有一个名为rc.local（或与此相似）的文件，这个文件将专门负责启动安装在本机器里的各种服务。在一个这样的系统上，把下面几条语句添加到rc.local文件里就能在系统开机时自动启动MySQL服务器：

```
if [ -x /usr/local/bin/mysqld_safe ]; then
    /usr/local/bin/mysqld_safe &
fi
```

如果你的MySQL bin目录与示例中的路径名不一样，请进行相应的修改。

在System V风格的系统上，可以安装mysql.server脚本——把它正确地拷贝到目录/etc下的启动目录里即可。如果你是在Linux系统上用RPM文件安装MySQL的，安装过程应该把这些事情都做好了。如果需要由你来安装mysql.server脚本，先把这个脚本用打算使用的名字拷贝到专门用来存放各种启动脚本的主启动目录里，再把这个脚本设置为可执行，然后在有关的运行级别目录里创建一个指向该脚本的符号链接即可。

**注意** 在以下的讨论中，将假设mysql.server脚本是以“mysql”为名被安装到启动目录里的，但为了让大家知道所指的是什么东西，将继续使用mysql.server来称呼它。

用来存放启动文件的目录在不同的系统上往往有着不同的布局结构，请大家根据自己系统的具体情况来研读下面给出的操作示例。比如说，在Solaris系统上，最常见的多用户运行级别是2，用来存放各种启动脚本的主启动目录通常是/etc/init.d，对应于运行级别2的目录则是/etc/rc2.d，所以有关命令将如下所示：



```
# cp mysql.server /etc/init.d/mysql
# cd /etc/init.d
# chmod +x mysql
# cd /etc/rc2.d
# ln -s ../init.d/mysql S99mysql
```

完成上述设置之后，系统的开机启动过程就能自动以start为参数去调用S99mysql脚本了。

Linux系统的启动目录集有着类似的布局结构，但它们通常都安排在/etc/rc.d目录下（比如/etc/rc.d/init.d和/etc/rc.d/rc3.d）。Linux系统大都带有一个专门用来管理各种启动脚本的chkconfig命令，可以用这个命令来帮助你安装mysql.server脚本，这样就用不着你亲自去敲入和执行上面那些命令了。下面是在某个Linux系统上把mysql.server脚本以“mysql”为名安装到启动目录的步骤：

1) 把mysql.server脚本拷贝到init.d目录里，然后把它设置为可执行：

```
# cp mysql.server /etc/rc.d/init.d/mysql
# chmod +x /etc/rc.d/init.d/mysql
```

2) 注册这个脚本程序并激活它：

```
# chkconfig --add mysql
# chkconfig mysql on
```

可以用chkconfig --list命令来检查一下这个脚本安装得是否正确，如下所示：

```
# chkconfig --list mysql
mysql          0:off  1:off  2:on   3:on   4:on   5:on   6:off
```

这个输出表明启动过程将在运行级别是3、4和5时自动执行mysql.server脚本。

如果没有chkconfig命令可用，可以参照Solaris系统上的有关步骤完成mysql.server脚本的安装工作，但要注意这两种系统在启动目录路径名方面的不同之处。以运行级别3为例，需要使用下列命令来安装mysql.server脚本：

```
# cp mysql.server /etc/rc.d/init.d/mysql
# cd /etc/rc.d/init.d
# chmod +x mysql
# cd /etc/rc.d/rc3.d
# ln -s ../init.d/mysql S99mysql
```

Mac OS X系统上的开机过程又有所不同。在Mac OS X系统上，将在系统开机过程中启动的各项服务通常都存放在目录/Library/StartupItems和/System/Library/StartupItems的下级子目录里。既可以参照某项现有服务的安装设置情况来安装MySQL，也可以简单地从网址<http://www.entropy.ch/software/macosex/mysql/>处下载一个用来启动MySQL的启动软件包。注意，在安装好这个软件包之后，可能还需要根据MySQL服务器在系统上的具体安装位置对该软件包中的主脚本做几处修改。

### 11.2.2 在Windows系统上运行MySQL服务器

用来在Windows系统上安装MySQL的发行版本通常都包括好几个服务器程序，这些服务器

程序是分别使用不同的选项编译出来的。本书的附录A对各种不同的MySQL服务器程序进行了介绍。在下面的讨论里，将在有关示例里分别以mysqld（适用于各种Windows版本）和mysqld-nt为例（适用于基于NT的Windows版本，例如，Windows NT、2000、XP等等）来介绍这些服务器程序的安装步骤。

Windows的任何版本都允许从命令行以手动方式启动MySQL服务器。除此之外，基于NT的系统还允许把任何一种MySQL服务器安装为一项服务——既可以把MySQL服务器设置成一项在Windows启动时自动运行的服务，也可以通过命令行或Windows的“Services Manager”（服务管理器）来控制它。如果你使用的是一个专为NT系统而编译出来的MySQL服务器，还可以把它设置成允许MySQL客户程序通过命名管道来建立连接。

### 1. 以手动方式运行MySQL服务器

在命令行上敲入下面这条命令就能以手动方式启动一个MySQL服务器：

```
C:\> mysqld
```

如果想让出错信息显示在控制台窗口里而不是被写入错误日志（即MySQL数据目录中的mysql.err文件），请加上一个--console选项，如下所示：

```
C:\> mysqld --console
```

在基于NT的系统上，如果打算允许MySQL客户程序通过一个命名管道来连接MySQL服务器，请使用mysqld-nt作为MySQL服务器程序。在MySQL 3.23.49版本之前，mysqld-nt中的命名管道支持机制是默认激活的；但在MySQL 3.23.50及以后的版本里，如果想激活对命名管道的支持，就必须明确地在启动命令里加上--enable-named-pipe选项。（注意：激活对命名管道的支持并不一定是件好事！人们之所以在后来的MySQL版本里默认禁用了对命名管道的支持，是因为发现它在很多机器上系统关机时会引起问题。如果要使用这个选项，请留意MySQL服务器的关停操作是否正常。）

如果想关停MySQL服务器，请使用mysqladmin程序：

```
C:\> mysqladmin -p -u root shutdown
```

### 2. 把MySQL服务器运行为一项服务

在基于NT的Windows版本上，可以用下面这条命令把MySQL服务器安装为一项服务：

```
C:\> mysqld-nt --install
```

此后，每当Windows启动时，MySQL服务器就会自动运行。但如果你更喜欢使用一项非自动运行的服务，也可以把MySQL服务器安装为一项需要以手动方式启动的服务：

```
C:\> mysqld-nt --install-manual
```

虽然这几个例子里使用的是mysqld-nt，但实际上能把任何一种MySQL服务器安装为一项服务。比如说，如果你不关心MySQL服务器能否支持命名管道，就完全可以安装mysqld。

通常，当把某个服务器安装为一项服务的时候，最好不要在命令行上列出它的选项——应该把它们列在某个选项文件里（请参见第11.2.3节）。但这个原则也有例外，即当打算把多个MySQL服务器都安装为同一台Windows机器上的服务时；这方面的详细情况请参见第11.6节。

把MySQL服务器安装为一项服务之后，就可以利用它的服务名“MySQL”通过命令行或者——如果你更喜欢图形化操作界面的话——Windows的“Services Manager”（服务管理器）来控制它了。根据具体使用的Windows版本，可以在Windows的“Control Panel”（控制面板）中的“Services”（服务）或“Administrative Tools”（管理工具）窗口里找到“Services Manager”（服务管理器）。

从命令行启动或停止MySQL服务要使用下面两条命令（服务名“MySQL”其实可以写成大小写字母的任意组合，因为Windows不区分名字中的字母大小写情况）：

```
C:\> net start MySQL
C:\> net stop MySQL
```

如果你使用的是“Services Manager”（服务管理器），Windows会把它知道的各项服务列成一份清单显示在一个窗口里，还可以在这个窗口里看到其他一些信息，比如某项服务是否正在运行、它是自动运行的还是手动运行的等等。从这个窗口所显示的清单里选中MySQL服务，再点击相应的按钮或菜单项就能启动或者关闭MySQL服务器了。

也可以在命令行上用mysqladmin shutdown命令来关停MySQL服务器。

如果想把MySQL服务器从Windows服务清单中去掉，需要先关停正在运行的MySQL服务器（如果它正在运行的话），然后发出下列命令：

```
C:\> mysqld-nt --remove
```

**注意** 虽说使用Windows的“Services Manager”（服务管理器）和DOS命令行都能对各种服务进行控制，但这两种办法最好不要同时交替使用；在DOS窗口里输入与服务有关的命令之前，应该先把“Services Manager”（服务管理器）窗口关掉。

### 11.2.3 设定MySQL服务器的启动选项

无论是在哪一种平台上，在启动MySQL服务器时都可以随意选用下面两种办法来给出启动选项：

- 在命令行上列出启动选项。既可以使用启动选项的短格式，也可以使用它们的长格式（如果它们有长格式的话）。比如说，既可以使用“--user=mysqladm”，也可以使用“-u mysqladm”。
- 把启动选项放到某个选项文件里去。此时，只能使用选项的长格式，并且选项名前面的两个连字符不要写出，如下所示：

```
user=mysqladm
```

一般说来，选项文件更便于使用。首先，它们既适用于手动方式，也适用于自动方式；其次，一旦设置好选项文件，它们就能在MySQL服务器每次启动的时候都发挥作用。在命令行上列出启动选项的做法只适用于以手动方式或通过mysqld\_safe脚本去启动MySQL服务器的情况（mysql.server脚本因为只支持start和stop两个命令行选项，所以不能用这个办法来启动）。此外，除将在第11.6节介绍的极少数例外情况外，当用--install或--install-manual把一个供Windows使用

的MySQL服务器安装为一项服务时，通常也不能在命令行上给出它的启动选项。

在UNIX系统上，用来设定MySQL服务器启动选项的常用文件是/etc/my.cnf文件和数据目录中的my.cnf文件。在Windows系统上，可以使用Windows系统目录中的my.ini文件、C:\my.cnf文件和MySQL数据目录中的my.cnf文件。如果要使用的文件不存在，请创建它。

一般说来，MySQL服务器的启动选项要放在[mysqld]选项组里。比如说，如果想让MySQL服务器运行在mysqladm用户名下并使用目录/usr/local/mysql作为它的基目录，就要在选项文件里写出以下内容：

```
[mysqld]
user=mysqladm
basedir=/usr/local/mysql
```

这相当于从命令行用以下选项来启动MySQL服务器：

```
% mysql --user=mysqladm --basedir=/usr/local/mysql
```

MySQL服务器和服务程序所使用的选项组的完整清单如下表所示：

程 序	所使用的选项组
mysqld	[mysqld]、[server]
mysqld_safe	[mysqld]、[server]、[mysqld_safe]、[safe_mysqld]
safe_mysqld	[mysqld]、[server]、[safe_mysqld]
mysql.server	[mysqld]、[mysql_server]
libmysqld	[embedded]、[server]

libmysqld是嵌入式MySQL服务器，可以把它链接到其他程序里以生成一个自成一体的应用程序，这类应用程序在运行时不需要在其外部独立运行的MySQL服务器的支持就能实现对数据库的访问。（本书的第6章对使用嵌入式MySQL服务器来开发应用程序的过程做了介绍。）

[server]选项组里的选项对独立式和嵌入式MySQL服务器都能起作用。[mysqld]或[embedded]选项组里的选项只能分别对独立式或嵌入式MySQL服务器起作用。类似地，[mysqld\_safe]或[mysql\_server]选项组里的选项只在使用了相应的启动脚本时才起作用。

mysqld\_safe脚本在MySQL 4之前的版本里叫做safe\_mysqld。本书对mysqld\_safe脚本和[mysqld\_safe]选项的讨论同样适用于safe\_mysqld脚本和[safe\_mysqld]选项。

MySQL选项文件的格式和语法详见本书的附录E。

如果是用一个启动脚本来启动MySQL服务器的，还可以考虑第三种MySQL启动选项的设定办法：把启动选项硬编码在启动脚本里，让启动脚本把有关选项直接传递给MySQL服务器程序。这个办法的最大缺点是你必须记住自己设置的每一个选项并在系统环境发生变化时，比如安装了一个MySQL新版本的时候（因为安装过程将把以前的启动脚本覆盖掉），重新对有关脚本做出一模一样的修改才行。

#### 11.2.4 关闭服务器

可以用mysqladmin程序手动地关闭MySQL服务器：



```
% mysqladmin -p -u root shutdown
```

这个办法对UNIX和Windows系统都适用。如果把MySQL服务器安装为Windows系统上的一项服务，还可以用下面这条命令来手动地关闭MySQL服务器：

```
C:\> net stop MySql
```

当然，也可以通过Windows的“Services Manager”（服务管理器）提供的图形化操作界面来选中并关闭MySQL服务器。

如果把MySQL服务器配置成在系统开机时自动启动，它在系统关机时将自动结束运行，用不着再对它的关闭过程做特殊配置。BSD UNIX系统大都通过向有关进程发一个TERM信号的办法来关闭某项服务；收到这个信号的进程或者在做出适当的响应（把内存里的数据写入磁盘，关闭已经打开的文件等）后“体面地”自行结束自己的运行，或者被这个信号粗暴而又悄无声息地取消掉。mysqld在收到这个信号时能够“体面地”结束自己的运行。

在System V风格的UNIX系统上，如果MySQL服务器当初是用mysql.server脚本启动的，关机过程将以stop为参数再次调用这个脚本去通知MySQL服务器结束运行。你也可以亲自执行这个脚本去手动地关闭MySQL服务器。比如说，假设把mysql.server脚本安装为/etc/rc.d/init.d/mysql，可以这样来调用它（这个操作必须以root用户身份执行）：

```
# /etc/rc.d/init.d/mysql stop
```

在基于Windows NT的系统上，如果把MySQL服务器运行作为一项服务，Windows的“Services Manager”（服务管理器）将在系统关机时自动通知MySQL服务器结束运行。在Windows的其他版本里，如果没有把MySQL服务器运行作为一项服务，就必须在退出Windows之前在命令行上用mysqladmin shutdown命令手动地结束MySQL服务器的运行。

### 11.2.5 在连接不上MySQL服务器时重新获得对服务器的控制

在某些场合，可能需要在连接不上MySQL服务器的情况下重新获得对它的控制并重新启动它。这句话有点儿自相矛盾，因为一般需要先用mysqladmin shutdown命令连接到MySQL服务器并通知它结束运行之后才能重新启动它。那么，这种矛盾的局面是如何发生的呢？

首先，MySQL的root口令有可能被意外地设置成了一个你不知道的值。这种失误往往发生在修改root口令的时候——比如说，在输入新口令时不小心碰到了某个控制键，当然，也可能是你根本就没记住口令。

其次，在UNIX系统上，本地客户以localhost为主机名建立的MySQL连接是通过一个UNIX套接字文件（比如说，/tmp/mysql.sock文件）实现的。如果这个套接字文件被删掉了，本地客户——因为这个连接已经不复存在——就将无法连接MySQL服务器。如果系统管理员（也许就是你本人）安排了一个cron作业去定期清理/tmp目录里的临时文件，就难免发生这样的意外。

如果无法连接MySQL服务器的原因是套接字文件被删掉了，那只要重新启动MySQL服务器就能把它给找回来，MySQL服务器将在再次启动时重新创建出套接字文件。问题是套接字文件已经不存在了，所以你将无法使用套接字来建立一条连接去通知MySQL服务器停止运行。此时，必须建立一条TCP/IP连接，即使用127.0.0.1而不是localhost作为--host（主机名/主机地址）参数



值去连接本地主机上的MySQL服务器:

```
% mysqladmin -p -u root -h 127.0.0.1 shutdown
```

127.0.0.1是一个IP地址值(本地主机的回馈地址),所以这条命令将强制性地建立起一条TCP/IP连接而不是建立起一条套接字连接。关闭MySQL服务器后,再重新以localhost为主机名启动之,它就会重新创建出一个套接字文件来。

不过,如果事情的确是因为套接字文件被某个cron作业删掉了才发生的话,不消除问题根源,找回来的套接字文件迟早还是会再次失踪的;必须修改那个cron作业或者使用一个位于其他地点的套接字文件。可以在某个全局选项文件里另行指定一个套接字文件。比如说,假设MySQL安装路径是/usr/local/mysql目录,在/etc/my.cnf文件里加上下面几条指令就能把套接字文件创建在这个目录里:

```
[mysqld]
socket=/usr/local/mysql/mysql.sock

[client]
socket=/usr/local/mysql/mysql.sock
```

为了让MySQL服务器和客户程序使用同一个套接字文件进行通信,在[mysqld]和[client]选项组里设定的套接字文件必须有相同的路径名。如果只在[mysqld]选项组里设定一个路径名,客户程序就会到老地方去寻找套接字。完成修改后重新启动MySQL服务器,它就将在新地点创建出一个套接字文件来。这个办法的不足之处是它只对那些会去读取选项文件的客户程序才有效;很多客户程序都会去读取选项文件,但也有些客户程序不这样做。如果从源代码开始重新编译MySQL,可以重新配置MySQL发行版本,让MySQL服务器和客户程序都默认使用同一个不同的路径名;这还将自动影响到那些使用MySQL客户程序开发库开发出来的第三程序。

如果你是因为忘记或者不知道root口令而无法连接MySQL服务器,就需要重新获得对MySQL服务器的控制才能重新设置root口令。此时,应该按以下步骤进行:

1) 关闭MySQL服务器。在UNIX系统上,如果能登录为MySQL服务器主机上的root用户,可以使用kill命令结束MySQL服务器的运行。先从MySQL服务器的PID文件(这个文件通常可以在MySQL数据目录里找到)或者使用ps命令查出MySQL服务器的进程ID,然后向MySQL服务器进程发出一个TERM信号让它结束运行:

```
# kill -TERM PID
```

MySQL服务器在收到TERM信号后会把内存中的数据表和日志信息写入磁盘。如果MySQL服务器因为种种原因而没有对这个正常终止信号做出响应,可用kill -9命令强行终止它:

```
# kill -9 PID
```

不过,不到万不得已,最好不要这样做;因为强行终止MySQL服务器进程往往会使MySQL来不及把内存中的信息写入磁盘而造成数据表里的数据不完整。

Linux系统上的ps命令可能会列出几个mysqld“进程”。这些“进程”其实只是同一MySQL服务器进程的多个线程,只要终止其中的任何一个,就能把它们全部终止。

如果你是用mysql\_safe脚本去启动MySQL服务器的，这个脚本将监控着MySQL服务器的运行情况并在它被终止时立刻重新启动之。此时，应该先查出mysqld\_safe进程的PID并终止mysqld\_safe进程，然后才能真正终止mysqld进程。

如果把MySQL服务器运行为Windows系统上的一项服务，那么，即便不知道它的口令，也能通过Windows的“Services Manager”（服务管理器）或者下面这条命令正常地结束它的运行：

```
C:\> net stop MySql
```

在Windows系统上，还可以通过“Task Manager”（任务管理器，Alt-Ctrl-Del组合键）来强行终止MySQL服务器的运行。这相当于UNIX系统上的kill -9命令，不到万不得已，最好不要这样做。

2) 用--skip\_grant\_tables选项重新启动MySQL服务器。此时，MySQL服务器将不使用它的权限表对连接操作进行身份验证，而你就能在不提供root口令的情况下连接上MySQL服务器并获得全部的权限了。但这等于是完全解除了MySQL服务器的安全防线并让别人也能以同样的方法连接上MySQL服务器，所以应该在连接上MySQL服务器之后尽快发出一条FLUSH PRIVILEGES语句：

```
% mysql
mysql> FLUSH PRIVILEGES;
```

这条FLUSH语句的作用是告诉MySQL服务器把权限表重新读入内存并开始对此后的连接请求进行身份验证。你仍将保持在连接状态，但MySQL服务器将像往常那样用这些权限表对此后到达的、来自其他客户的连接请求进行身份验证。这条FLUSH语句还重新激活了GRANT语句；在MySQL服务器不使用权限表时，GRANT语句是被禁用的。重新加载权限表后，就可以按照第11.1节中给出的有关步骤去修改root口令了。

3) 修改完root口令后，应该关闭MySQL服务器并按正常启动过程再次启动之。

如果你当初是用UNIX命令kill -9或Windows的“Task manager”（任务管理器）去强行终止MySQL服务器的运行的，这种“简单粗暴”的关停操作会让MySQL服务器来不及把未存盘的信息写入磁盘，这往往会导致某些数据表遭到破坏。为避免这种粗暴的关停操作可能导致的问题，应该提前激活MySQL服务器的自动恢复机制；有关这方面的详细讨论请参见本书的第13章。

### 11.3 管理MySQL用户账户

MySQL管理员应该知道如何设置MySQL用户账户，即指定哪些用户可以连接到MySQL服务器、他们都可以从哪些地方去连接MySQL服务器、连接上MySQL服务器后又能做些什么等等。这些信息分别存放在mysql数据库中的各个权限表里，并主要通过下面两条语句来管理：

- GRANT：创建MySQL账户并设定它的权限。
- REVOKE：收回现有MySQL账户的权限。

这两条语句最早出现于MySQL 3.22.11版本，是人们为简化用户账户的管理工作而引入的。在MySQL 3.22.11之前的版本里，只能通过INSERT和UPDATE等SQL语句来直接处理权限表的内容。GRANT和REVOKE语句相当于权限表的一个操作前端，它们在概念上更清晰，用起来也

更容易；在你对想要分配的权限做出描述之后，MySQL服务器将自动地把你的请求映射为正确的权限表操作。不过，虽说使用GRANT和REVOKE语句的做法要比直接修改权限表的做法更简明，但仍建议大家把第12章的内容作为这一节的补充材料。第12章对权限表所做的详细介绍能够帮助大家更好地理解和掌握GRANT和REVOKE语句的工作原理。在第12章里，还专门安排了一个小节来介绍如何在不使用GRANT语句的前提下设置MySQL账户——如果你的MySQL服务器早于3.22.11版本，就需要按照那一小节所介绍的步骤去设置MySQL账户的权限。

此外，也可以使用附带在MySQL发行版本中的mysqlaccess和mysql\_setpermission脚本来设置MySQL账户的权限。这两个脚本都是用Perl语言编写出来的，它们向我们提供了一种可以代替GRANT语句去设置MySQL用户账户的手段。mysql\_setpermission脚本必须在机器里安装有DBI支持时才能使用。

GRANT和REVOKE语句会影响到四个权限表，如下所示：

权限表的名称	权限表的内容
user	允许连接到MySQL服务器的用户以及它们的全局级权限
db	数据库级权限
tables_priv	数据表级权限
columns_priv	数据列级权限

第五个权限表的名字是host，但因为它不受GRANT或REVOKE语句的影响，所以把它安排到第12章再做介绍，这里就不讨论了。

每当用GRANT语句创建一个账户时，user权限表就将相应地增加一条关于这个账户的记录项；如果所发出的GRANT语句还为此账户设定了全局级权限（管理性操作权限或者使用该用户能够对所有的数据库进行某种操作的权限），它们也将被记录到user权限表里。如果所设定的权限只适用于某个给定的数据库、数据表或数据列，它们将被分别记录到db、tables\_priv和columns\_priv权限表里去。

在以下内容里，将向大家依次介绍如何创建MySQL用户账户并进行授权、如何收回权限和从权限表里彻底删除某个用户以及如何改变口令或重新设置丢失的口令。

### 11.3.1 创建MySQL用户账户并进行授权

GRANT语句的语法如下所示：

```
GRANT privileges (columns)
ON what
TO account IDENTIFIED BY 'password'
REQUIRE encryption requirements
WITH grant or resource management options;
```

在GRANT语句的子句当中，有几个是可选的；如果用不着，就不必把它们写出来。下面是GRANT语句最为常用的几个子句：

- **privileges** 授予有关账户的权限。比如说，SELECT权限允许用户发出SELECT语句，SHUTDOWN权限允许用户关停MySQL服务器。

- **columns** 有关权限将作用于哪些数据列上。这个子句是可选的，只有在需要设置数据列级权限时才必须给出这条子句。如果需要列举多个数据列，就要把它们的名字用逗号隔开。
- **what** 有关权限的级别。最高级别是全局级，即给定权限将作用于所有的数据库和所有的数据表；可以把全局级权限视为超级用户权限。有关权限还可以被设定为数据库级、数据表级或者（当还给出了一条**columns**子句时）数据列级。
- **account** 被授予有关权限的账户。**account**值由以'**user\_name**'@'**host\_name**'格式给出的一个用户名和一个主机名构成，因为MySQL不仅要求指定谁能连接，而且还得指定他从何处进行连接。这意味着即便两个用户有着相同的名字，如果他们将从不同地点去连接MySQL服务器的话，就要为他们两人各自创建一个账户并各自授予相应的权限；而在你做出这样的设置之后，MySQL也知道该如何区别对待这两位用户。某账户的**user\_name**和**host\_name**值将被记录在该账户在**user**权限表里的记录项的**User**和**Host**数据列里，这两个值还将被记录在相应的**GRANT**语句在其他权限表里为这个账户所创建的各有关记录里。MySQL用户名是在连接MySQL服务器时用来表明自己身份的名字，它与UNIX登录名或者Windows登录名没有任何关系。在默认的情况下，如果在连接MySQL服务器时没有明确地给出一个MySQL用户名，用来连接MySQL服务器的那个MySQL客户程序就将使用登录名作为MySQL用户名，但这只是一种约定。类似地，MySQL把“root”用做其超级用户的名字也不是出于什么特殊的考虑，这也是一种约定。完全可以把权限表里的这个名字改为“nobody”，然后再以nobody为用户名去连接MySQL服务器并去进行各种必须具备超级用户权限才能进行的操作。
- **password** 有关账户的口令。这个子句是可选的；如果在创建新账户时没有使用**IDENTIFIED BY**子句，新创建出来的账户将没有口令（这可是个安全漏洞）。如果在使用**GRANT**语句修改某个老账户的权限时没有使用**IDENTIFIED BY**子句，该账户的口令将保持不变；如果给出了**IDENTIFIED BY**子句，该账户的口令将被替换为新给出的口令。**IDENTIFIED BY**子句中的**password**值是口令的明文形式，**GRANT**语句将自动对之进行加密——千万不要像使用**SET PASSWORD**语句那样用**PASSWORD()**函数对口令进行加密。

**REQUIRE**和**WITH**子句是可选的。**REQUIRE**子句最早出现于MySQL 4.0.0版本，其用途是对必须经由SSL进行加密连接的账户进行设置。**WITH**子句用来授予**GRANT OPTION**权限，这个权限允许有关账户把它自己的权限再转授给其他用户。从MySQL 4.0.2版本开始，**WITH**子句还可以用来设置资源管理选项，即对有关账户每小时允许进行多少次连接或最多进行多少次查询的最大次数进行设置；这些选项有助于避免某些账户独占MySQL服务器。

在权限表里，用户名、口令、数据库名、数据表名等字段区分字母的大小写情况，主机名和数据列名字段不区分字母的大小写情况。

在创建账户的时候，可以通过回答下面这几个简单的问题来构造出相应的**GRANT**语句：

- 谁将从何处建立连接？该用户的用户名是什么？该用户将从哪台主机开始连接MySQL服务器？
- 这个账户将用来执行哪些访问操作？或者说这位用户应该拥有什么权限级别？这些权限又将作用于哪些东西？



- 是否需要使用加密连接?
- 是否要向这位用户授予一些管理员权限?
- 是否需要限制这位用户的资源占用量?

下面,将通过一些示例来回答这些问题并构造出相应的GRANT语句,希望这些示例能帮助大家熟悉MySQL用户账户的创建和设置过程。

#### 1. 谁将从何处建立连接

GRANT语句的account部分负责设定:1)MySQL用户的用户名;2)该用户都允许从什么地方去连接MySQL服务器。作为MySQL管理员,需要根据具体情况对允许每位用户使用哪几台计算机去连接MySQL服务器的事情做出设定。作为一种极端的情况,可以把允许某位用户用来连接MySQL服务器的计算机限制为只有一台,让他只能使用那一台计算机去访问数据库。比如说,下面两条语句都是只允许某给定用户从某一台给定主机去访问sampdb数据库里的所有数据表:

```
GRANT ALL ON sampdb.* TO 'boris'@'localhost' IDENTIFIED BY 'ruby';
GRANT ALL ON sampdb.* TO 'fred'@'ares.mars.net' IDENTIFIED BY 'quartz';
```

如果account值的用户名或主机名部分不包含“-”或“%”等特殊字符,就不必用单引号把它们引起来(比如说,不带引号的boris@localhost是合法的)。但为了避免不必要的麻烦,还是使用引号为好;本书中的示例都把这当做一条规则来遵守。特别提醒大家注意的是,用户名和主机名必须分别放在两对单引号里,即必须写成'boris'@'localhost',不能写成'boris@localhost'。

只允许某位用户从某一台主机去连接MySQL服务器是最为严格的访问控制形式。在另一种极端的情况下,公司可能会有一位因经常出差而需要从全球各地许多不同的主机连接MySQL服务器的用户。假设这位用户的用户名是max,下面这条语句将允许他从任何地方连接MySQL服务器:

```
GRANT ALL ON sampdb.* TO 'max'@'%' IDENTIFIED BY 'diamond';
```

“%”字符是一个通配符,它在这里的含义与它在LIKE模式匹配操作中的含义相同。因此,把主机名设置为“%”就表示“任意一台主机”。这种设置对用户最为方便,但对系统却最不安全。(这种设置往往会给你带来很多令人头疼的问题,我们将在第12章对此做进一步探讨。)

在两个极端之间,通常需要允许某位用户从一组有限的主机去连接MySQL服务器。比如说,如果能让用户mary能够从snake.net域中的任意一台主机去连接MySQL服务器,就需要把主机名设置为%.snake.net:

```
GRANT ALL ON sampdb.* TO 'mary'@'%.snake.net' IDENTIFIED BY 'topaz';
```

用来匹配任一字符的另一个LIKE通配符(“\_”)也可以用在主机名部分里。

account值的主机部分还允许以IP地址而不是主机名的形式给出,IP地址既可以完全是数字,也可以包含模式匹配字符。此外,在MySQL 3.23及以后的版本里,还可以利用网络掩码来表明IP地址中的哪些位是有效的,如下所示:

```
GRANT ALL ON sampdb.* TO 'joe'@'192.168.128.3' IDENTIFIED BY 'water';
GRANT ALL ON sampdb.* TO 'ardis'@'192.168.128.%' IDENTIFIED BY 'snow';
GRANT ALL ON sampdb.* TO 'rex'@'192.168.128.0/255.255.128.0'
IDENTIFIED BY 'ice';
```



上面第一条语句表明用户joe只能从指定主机去连接MySQL服务器。第二条语句给出的是C类子网192.168.128中的一个IP地址范围。在第三条语句里, 192.168.128.0/255.255.128.0中的掩码表明IP地址中的前17位是有效的, 即允许用户从其IP地址的前17位是192.168.128的任何一台主机去连接MySQL服务器。

在GRANT语句里使用localhost作为主机名将允许该用户以localhost或127.0.0.1(本地主机的回馈IP地址)为主机名去连接运行在本地主机上的MySQL服务器。在支持命名管道的Windows系统上, 以localhost作为主机名的账户还将允许该用户以“.”(句点)为主机名去连接MySQL服务器。在UNIX系统上, 以localhost为主机名的连接将通过一个UNIX套接字文件建立起来。在支持命名管道的Windows系统上, 以“.”(句点)为主机名的连接将通过一个命名管道建立起来(在不支持命名管道的Windows系统上, 以“.”为主机名将建立起一条TCP/IP连接)。所有其他的连接都将通过TCP/IP建立, 以本地主机回馈地址127.0.0.1作为主机名部分的连接也将是一条TCP/IP连接。

如果在创建某个账户的时候没有给出它的主机名部分, 在效果上就与把主机名部分设置为“%”完全相同。也就是说, 在GRANT语句里把account值设置为'max'或'max'@'%'是等效的。这进一步意味着如果你本想把某个账户设置为'boris'@'localhost'却误写成了'boris@localhost', 它仍将被MySQL接受为一个合法的账户——MySQL将把'boris@localhost'整个地解释为该账户的用户名部分并给它加上一个'%'作为其默认的主机名部分, 最终得到的账户名将是'boris@localhost'@'%'。要想避免这种错误, 就一定要给账户名中的用户名部分和主机名部分各自加上单引号。

#### 如何在权限表的记录项里写出本地主机名

在运行着MySQL服务器的主机上, 使用该主机的完整主机名而不是localhost去连接MySQL服务器经常会遇到一些麻烦。这往往是因为在权限表里写出的主机名与域名解析器所报告出来的主机名不一致而造成的。假设MySQL服务器主机的完整主机名是cobra.snake.net。那么, 如果域名解析器报告出来的是一个不完整主机名(如cobra)而权限表里写的是它的完整名称(或正好相反), 就会造成不匹配。

如果想知道在系统上会不会发生这种问题, 可以在MySQL服务器主机上使用-h选项和这台主机的完整主机名连接一下MySQL服务器试试, 比如:

```
% mysql -h cobra.snake.net
```

然后查看一下MySQL服务器的常规日志文件。常规日志是如何记载这次连接的? 其中写的是本地主机的完整主机名还是一个不完整的主机名? 常规日志里记载的本地主机名是什么格式, 在GRANT语句里就应该使用什么格式来设定有关账户的本地主机名部分。

#### 2. 应该给用户授予哪些访问权限

可授予用户的访问权限有很多种。我们把这些访问权限归纳在表11-1里, 并将在第12章对它们的用途以及它们与各权限表的关系做详细的介绍。

表11-1 MySQL的访问权限

访问权限	该权限所允许的操作
CREATE TEMPORARY TABLES	创建临时数据表
EXECUTE	执行存储过程（保留供今后使用）
FILE	读、写MySQL服务器主机上的文件
GRANT OPTION	把本账户的权限授予其他账户
LOCK TABLES	用LOCK TABLES语句明确地锁定某个特定的数据表
PROCESS	查看关于在MySQL服务器里运行着的各有关线程的信息
RELOAD	重新加载各种权限表或者对各有关日志及缓存区进行刷新
REPLICATION CLIENT	查知主/从MySQL服务器的运行地点（即主/从MySQL服务器所在的主机名）
REPLICATION SLAVE	运行为一个镜像从MySQL服务器
SHOW DATABASES	发出SHOW DATABASES语句
SHUTDOWN	关闭MySQL服务器
SUPER	用KILL命令终止线程以及进行其他超级用户操作
ALTER	更改数据表和索引的定义
CREATE	创建数据库和数据表
DELETE	删除数据表里的现有数据行
DROP	丢弃（删除）数据表和数据行
INDEX	创建或者丢弃索引
INSERT	往数据表里插入新数据行
REFERENCES	未使用（保留供今后使用）
SELECT	对数据表里的现有数据行进行检索
UPDATE	对数据表里的现有数据行进行修改
ALL	所有操作（但不包括GRANT权限）；与ALL PRIVILEGES是同义词
USAGE	一个特殊的“无权限”权限

表中的第一组权限是管理权限。这类权限控制着MySQL服务器的运行情况，所以很少被授予普通用户。（比如说，用来关闭MySQL服务器的SHUTDOWN权限肯定不应该授予普通用户。）表中的第二组权限作用于数据库、数据表和数据列，它们控制着用户对MySQL服务器所管理的数据的访问操作。表中的第三组权限有着特殊的用途：ALL代表“所有权限”（但不包括GRANT OPTION权限）；USAGE代表“无权限”——意思是“创建一个账户，但不给它授予任何权限”。USAGE权限的主要用途是在不改变有关账户访问权限的前提下去修改该账户与访问权限没有关系的项（比如用户名、主机名等等）。

有些权限是从MySQL 4.0.2版本开始才新增加的；如果你的MySQL版本较早，就不能把这些权限用在权限授予操作中。这类权限包括CREATE TEMPORARY TABLES、EXECUTE、LOCK TABLES、REPLICATION CLIENT、REPLICATION SLAVE、SHOW DATABASES和SUPER。这类权限的引入改变了某些权限操作以往的控制方式。比如说，MySQL 4.0.2之前的版本里，要想结束在MySQL服务器里运行的任何线程，必须具备PROCESS权限；但在4.0.2及以后的版本里，却必须具备SUPER权限。如果在把MySQL服务器从4.0.2之前的某个版本升级到4.0.2或更高的版本时对权限表也进行了升级，这通常不成问题；MySQL发行版本中的mysql\_fix\_privilege\_tables脚本会为那些新权限在权限表里相应地增加一些数据列并把各有关账

户中的SUPER权限初始化为PROCESS权限。这样，能够在4.0.2之前的版本里结束有关线程运行的每一位用户在4.0.2及以后的版本里仍能这样做。（在升级到4.0.2或更高版本时用来对有关权限做出调整的具体步骤见第12.2.1节里的叙述。）

注意，在MySQL 4.0.2或4.0.3版本里，不能把具体针对某一个数据库的CREATE TEMPORARY TABLES或LOCK TABLES权限用在权限授予操作中，所以大家应该尽量避免使用这两个版本。这个问题在MySQL 4.0.4版本里得到了纠正。

MySQL允许在数据库系统全局、数据库、数据表、数据列等多个级别上进行授权。权限的级别由ON子句负责设定，如下表所示：

权限限定符	有关权限的作用范围
ON *.*	全局级权限，其作用范围是所有数据库里的所有数据表
ON *	全局级权限，若未指定默认数据库，其作用范围是所有数据库里的所有数据表；否则，其作用范围是当前数据库里的所有数据表
ON db_name.*	数据库级权限，其作用范围是指定数据库里的所有数据表
ON db_name.tbl_name	数据表级权限，其作用范围是指定数据表里的所有数据列
ON tbl_name	数据表级权限，其作用范围是默认数据库中指定数据表里的所有数据列

全局级权限的作用范围最大，它们允许你对任何一个数据库进行有关的操作。比如说，下面这条语句将使ethel用户成为一名能做任何事情（包括能对其他用户进行授权）的超级用户：

```
GRANT ALL ON *.* TO 'ethel'@'localhost' IDENTIFIED BY 'coffee'
WITH GRANT OPTION;
```

ON \*.\*子句的意思是“所有的数据库，所有的数据表”。出于安全方面的考虑，上面这条语句规定ethel用户只能从本地主机去连接MySQL服务器。限制MySQL超级用户只能从某几台主机去连接MySQL服务器是一种极其明智的做法——首先，要想盗取MySQL超级用户的口令，黑客们只能从这几台主机入手；其次，一旦你发现有人盗取了MySQL超级用户的口令，也只需从这几台主机入手去追踪黑客。

有些权限是专为数据库系统的管理员准备的，它们也只能通过全局级权限限定符ON \*.\*去进行授予。这类权限包括FILE、PROCESS、RELOAD、SHUTDOWN以及表11-1的第一组中的其他权限。比如说，RELOAD权限允许使用FLUSH语句；下面这条语句将创建出名为“flush”的用户，这位用户除了能发出FLUSH语句外其他什么事情都不能做：

```
GRANT RELOAD ON *.* TO 'flush'@'localhost' IDENTIFIED BY 'flushpass';
```

这类MySQL账户特别适合用来编写一些需要完成某种管理性操作的脚本。比如说，在日志文件的旋转过程中，需要把内存中的日志信息写到磁盘文件里去，而用前面创建的flush用户来做这件事将是最合适的（请参见第11.4.6节）。

数据库级权限将作用于给定数据库中的所有数据表。这类权限要用ON db\_name.\*子句来授予：

```
GRANT ALL ON sampdb.* TO 'bill'@'racer.snake.net' IDENTIFIED BY 'rock';
GRANT SELECT ON menagerie.* TO 'reader'@'%' IDENTIFIED BY 'dirt';
```

上面的第一条语句创建了一个名为“bill”的用户并把sampdb数据库中所有数据表上的所有权限都授予了这位用户。第二条语句创建了一个名为“reader”的用户，这位用户有权访问menagerie数据库里的任何一个数据表，但只能通过SELECT语句访问。换句话说，reader是一个“只读”用户。

MySQL允许在GRANT语句里同时列出多个权限，但要用逗号把它们彼此隔开。比如说，如果能让某位用户能读取和修改sampdb数据库里的现有数据表的内容，但不能创建新数据表或丢弃现有数据表，就不能把ALL权限授予给这位用户，应该像下面这样做：

```
GRANT SELECT,INSERT,DELETE,UPDATE ON sampdb.* TO 'jennie'@'%'
IDENTIFIED BY 'boron';
```

要想在数据库级别之下实现更细致的访问控制，就要在各有关数据表甚至是在各有关数据列上进行授权操作。给定一个数据表，如果能让其中的某些数据列对某位用户不可见，或者想让这位用户只能修改其中的某几个数据列，就需要用到数据列级权限。比如说，假设你是“美国历史研究会”的秘书，现在有一个志愿者要来帮你做些杂事。这是件好事，但因为member数据表里存放的都是会员们的个人资料，所以你决定开始时只把member数据表上的只读权限和该数据表的expiration数据列上的UPDATE权限授予给这位新助手。这样，当某位会员申请延长其会员资格时，你就可以放心地让这位新助手去修改那位会员的会员资格失效日期了。于是，你用下面两条语句为这位新助手创建了一个MySQL账户：

```
GRANT SELECT ON sampdb.member
TO 'assistant'@'localhost' IDENTIFIED BY 'officehelp';
GRANT UPDATE (expiration) ON sampdb.member
TO 'assistant'@'localhost';
```

第一条语句把对整个member数据表的读权限授予了新助手并设置了一个口令。第二条语句给他增加了UPDATE权限，但其作用范围仅限于expiration数据列。又因为第一条语句已经设置了一个口令，所以在第二条语句里就不必再设定一个口令了。

如果能让数据列级权限作用于多个数据列，就要以逗号为分隔符把有关数据列全都列举出来。比如说，如果能把member数据表的address数据列上的UPDATE权限也授予给assistant用户，可以再发出一条如下所示的语句，新权限将被追加到assistant用户已经拥有的权限清单里：

```
GRANT UPDATE (street,city,state,zip) ON sampdb.member
TO 'assistant'@'localhost';
```

权限表里的记录项不“追随”更名操作，例如，如果更改了某个数据表的名字，关联在该数据表上的所有权限将全部失效。这一原则适用于数据库级、数据表级和数据列级的各种权限。

#### “无权限”的USAGE权限有什么用

USAGE是一种含义为“无权限”的特殊权限。初看起来，它好像没有什么用，其实不然。当你想在继续保持其现有权限的情况下去修改某个账户与权限没有关系的项（比如用户名、主机名等等）时，就必须求助于USAGE权限。相应的GRANT语句的具体写法是：授予全局级USAGE权限，写出账户名，给出该账户与权限无关的新设置值。



比如说, 如果想在保持某账户现有权限的前提下改变该账户的口令、要求该账户的用户必须使用SSL连接或者想对该账户占用的系统资源加以限制的时候, 就要使用下面几条语句:

```
GRANT USAGE ON *.* TO account IDENTIFIED BY 'new_password';
GRANT USAGE ON *.* TO account REQUIRE SSL;
GRANT USAGE ON *.* TO account WITH MAX_CONNECTIONS_PER_HOUR 10;
```

### 3. 是否需要使用加密连接

从MySQL 4开始, MySQL允许使用SSL(安全套接字层)协议来建立一条加密连接, 它将对MySQL服务器和客户程序之间的数据流进行加密, 使它们不再以明文的形式传输。此外, 还可以把X509当做一种让客户程序在SSL连接上提供身份验证信息的手段来使用。加密连接给信息增加了一层安全屏障, 但因为需要进行加密/解密运算, 所以会加重CPU的负担。目前, 只有运行在UNIX系统上的MySQL版本才支持SSL。

与加密连接有关的选项要用REQUIRE子句来设置。REQUIRE SSL表示要求该用户必须通过SSL来连接MySQL服务器, 但对具体使用的加密连接类型不做具体要求:

```
GRANT ALL ON sampdb.* TO 'eladio'@'%.snake.net' IDENTIFIED BY 'flint'
REQUIRE SSL;
```

更具体些, 可以要求客户方提供一份有效的X509证书:

```
GRANT ALL ON sampdb.* TO 'eladio'@'%.snake.net' IDENTIFIED BY 'flint'
REQUIRE X509;
```

要是再具体些, REQUIRE子句还允许要求客户方提供的X509证书必须具备某些特征或者要求连接必须使用某种特定的密码类型来加密。这些特征由REQUIRE子句中的ISSUER、SUBJECT或CIPHER选项设定。(ISSUER和SUBJECT分别代表证书的签发者和持有者)。比如说, 在sampdb发行版本中的SSL目录里有一份客户身份证书文件client-cert.pem, 可以用这份文件来测试SSL连接。这份证书的签发者和持有者可以用下面这条命令查出来:

```
% openssl x509 -subject -noout -in client-cert.pem
issuer= /C=US/ST=WI/L=Madison/O=sampdb/OU=CA/CN=sampdb
subject= /C=US/ST=WI/L=Madison/O=sampdb/OU=client/CN=sampdb
```

下面这条GRANT语句所创建的账户要求客户方提供的证书必须匹配刚查出来的签发者和持有者:

```
GRANT ALL ON sampdb.* TO 'eladio'@'%.snake.net' IDENTIFIED BY 'flint'
REQUIRE ISSUER '/C=US/ST=WI/L=Madison/O=sampdb/OU=CA/CN=sampdb'
AND SUBJECT '/C=US/ST=WI/L=Madison/O=sampdb/OU=client/CN=sampdb';
```

要想明确地表明无需使用加密连接, 可使用REQUIRE NONE子句。NONE选项最早出现于MySQL 4.0.4版本。

在使用REQUIRE子句的时候, 还需要注意以下几个问题:

- 发出一条要求某账户必须使用加密连接的GRANT语句仅表明你对这个账户做出了一项限



制，可这并不能让人们使用该账户建立的连接自动地成为一条加密连接。要想建立加密连接，就必须：1）配置MySQL，使之支持SSL；2）以某种特定方式启动MySQL服务器和客户程序。有关的具体做法请参见第12章。

- 只要把SSL支持组件配置到了MySQL服务器与客户程序里，任何一位用户都可以去建立加密连接。REQUIRE子句只是一种用来规定某个账户必须使用加密连接去连接MySQL服务器的手段而已。
- 如果某个账户在连接MySQL服务器时不需要经过外部网络，用REQUIRE子句来要求该账户必须建立一条加密连接的做法就没有实际的意义。这类连接不可能被窃听，所以要求它们必须是加密连接只会毫无必要地加重CPU的计算负担而不会让你得到任何有实际意义的好处。在连接MySQL服务器时不需要经过外部网络的账户包括：1）只通过一个UNIX套接字文件去连接MySQL服务器的账户；2）只通过一个命名管道去连接MySQL服务器的账户；3）只连接到IP地址127.0.0.1（主机回环接口）的账户。这些连接所经过的路由全部包括在本地主机的内部，信息只在本地主机内部传输，根本不会泄露到外部网络上去。

#### 4. 是否要向用户授予一些管理员权限

一般说来，某数据库的所有者应该拥有该数据库上的全部访问控制，这就需要MySQL管理员把该数据库上的全部权限（包括WITH GRANT OPTION权限）授予给这位用户。比如说，如果想让用户alicia能够从big-corp.com域中的任何一台主机去连接MySQL服务器并授予其sales数据库中所有数据表上的管理权限，就需要发出一条如下所示的GRANT语句：

```
GRANT ALL ON sales.*
  TO 'alicia'@'%big-corp.com' IDENTIFIED BY 'shale'
  WITH GRANT OPTION;
```

实际上，WITH GRANT OPTION子句相当于把访问权限的授权权授予给指定用户，拥有GRANT OPTION权限的用户能够把自己的权限再转授给其他的用户。比如说，假设只向用户A授予了SELECT权限，但向用户B授予了GRANT OPTION权限、SELECT权限和其他几种权限，那么用户B——如果他愿意把自己的权限转授给用户A的话——就能使用户A变得更“强大”。这里要特别提醒大家注意的是，如果两位用户都拥有GRANT OPTION权限，他们就能把自己的权限相互转授给对方。

GRANT OPTION权限的另一种授予方法是把它直接写在GRANT语句的开头部分，如下所示：

```
GRANT GRANT OPTION ON sales.* TO 'alicia'@'%big-corp.com';
```

但下面这条语句却是错误的：

```
GRANT ALL, GRANT OPTION ON sales.* TO 'alicia'@'%big-corp.com';
```

ALL只能单独出现，不能与其他权限列举在一起。

#### 5. 是否需要限制用户的资源占用量

从MySQL 4.0.2版本开始，可以给MySQL用户每小时能连接到MySQL服务器的次数以及每小时能进行的查询或修改次数设置一个上限。这些上限值要用WITH子句来设置。请看下面这条GRANT语句，它把sampdb数据库上的全部权限都授予了spike用户，但只允许他每小时最多

连接10次、每小时最多发出200条查询命令（在这200条查询命令中，最多只能有50条是数据修改命令）：

```
GRANT ALL ON sampdb.* TO 'spike'@'localhost' IDENTIFIED BY 'pyrite'
    WITH MAX_CONNECTIONS_PER_HOUR 10 MAX_QUERIES_PER_HOUR 200
    MAX_UPDATES_PER_HOUR 50;
```

资源管理选项在WITH子句中的次序无关紧要。这些选项的默认值都是零，意思是“没有上限”。

如果某位用户具备RELOAD权限，他就能通过发出一条FLUSH USER\_RESOURCES语句的办法对资源管理选项的当前计数值进行复位；FLUSH PRIVILEGES语句也能做到这一点。在资源管理选项的当前计数值被复位之后，那些资源占用量已经达到其每小时上限值的账户就能再次进行连接和发出查询了。

### 11.3.2 收回权限和删除用户

要收回某位用户的权限，可使用REVOKE语句。除了把TO换成了FROM以外，REVOKE语句与GRANT语句在语法上非常相似；但REVOKE语句没有IDENTIFIED BY、REQUIRE和WITH子句：

```
REVOKE privileges (columns) ON what FROM account;
```

*account*部分必须与当初向这个账户授予权限时使用的GRANT语句的*account*部分相匹配。*privileges*部分则不必匹配；在用GRANT语句授予权限后，可以只用REVOKE语句收回其中的一部分权限。比如说，下面的GRANT语句授予了sampdb数据库上的全部权限，但REVOKE语句只收回了该账户对现有记录进行删除和修改的权限：

```
GRANT ALL ON sampdb.* TO 'boris'@'localhost' IDENTIFIED BY 'ruby';
REVOKE DELETE,UPDATE ON sampdb.* FROM 'boris'@'localhost';
```

GRANT OPTION权限不包括在ALL权限中，所以如果在授予该权限后又想收回它，就只能在REVOKE语句的*privileges*部分里把它明确地写出来：

```
REVOKE GRANT OPTION ON sales.* FROM 'alicia'@'%.big-corp.com';
```

REVOKE语句会收回某个账户的权限，但不会删除这个账户——哪怕已经收回了这个账户的全部权限，在user权限表里也仍留有一个与之对应的记录项。这意味着这个用户仍能连接MySQL服务器。要想真正删除某个账户，必须明确地使用DELETE语句把它在user权限表里的记录项删掉。比如说，如果真的想删除mary@%.snake.net账户，就需要像下面这样做：

```
% mysql -u root
mysql> USE mysql;
mysql> DELETE FROM user
    -> WHERE User = 'mary' and Host = '%.snake.net';
mysql> FLUSH PRIVILEGES;
```

DELETE语句负责从user权限表里删除对应于这个账户的记录项，FLUSH语句负责通知MySQL服务器重新加载权限表。这条FLUSH语句是必不可少的，这是因为当你使用GRANT或

REVOKE语句的时候，MySQL服务器将自动重新加载权限表；但当你直接修改权限表的时候，它却不会这样做。（要想彻底清除某个账户，还应该检查一下其他权限表里是否还有与它有关的记录项并在发出FLUSH PRIVILEGES语句之前把它们全都删掉。）

让人觉得奇怪的是，有些权限必须使用GRANT语句才能收回。比如说，MySQL允许你要求某个账户必须使用SSL去进行连接，却没有提供用来撤销这项要求的REVOKE语法，所以只能用一条如下所示的GRANT语句来做这件事：在全局级授予USAGE权限（保留该账户的现有权限），再用REQUIRE NONE子句表明今后不再要求它使用SSL来进行连接：

```
GRANT USAGE ON *.* TO account REQUIRE NONE;
```

类似地，用GRANT语句设置的资源占用量上限也无法用REVOKE语句来清除，只能用一条如下所示的GRANT语句来做这件事：在全局级授予USAGE权限（保留该账户的现有权限），再用WITH子句把有关的资源占用量上限值设置为零（即“没有上限”）：

```
GRANT USAGE ON *.* TO account
    WITH MAX_CONNECTIONS_PER_HOUR 0 MAX_QUERIES_PER_HOUR 0
    MAX_UPDATES_PER_HOUR 0;
```

### 11.3.3 修改口令或重新设置丢失的口令

修改或重新设置某账户口令的第一种办法是使用UPDATE语句——利用主机名Host和用户名User值从user权限表里查出有关账户的记录项并修改它的Password（口令）值，如下所示：

```
mysql> UPDATE user SET Password=PASSWORD('silicon')
    -> WHERE User='boris' AND Host='localhost';
mysql> FLUSH PRIVILEGES;
```

第二种办法是使用SET PASSWORD语句，比第一种办法要简便得多，账户名按GRANT语句中的简单格式写出即可，权限表也用不着去重新加载了：

```
mysql> SET PASSWORD FOR 'boris'@'localhost' = PASSWORD('silicon');
```

SET PASSWORD语句比UPDATE语句也更安全些，UPDATE语句比较容易写错和改错user权限表里的记录项。

修改口令的第三种办法是使用GRANT USAGE ... IDENTIFIED BY语句。这里要提醒大家注意的是：直接写出口令文本即可，千万不要画蛇添足地使用PASSWORD()函数：

```
mysql> GRANT USAGE ON *.* TO 'boris'@'localhost' IDENTIFIED BY 'silicon';
```

当你因为忘了root口令而无法连接MySQL服务器的时候，就需要重新设置root口令。这就产生了这样一个矛盾：应该先用root口令连接上MySQL服务器才能去修改root口令。如果不知道root口令，就只能先强行关停MySQL服务器，然后在不使用权限表来进行身份验证的情况下重新启动MySQL服务器；这一过程的具体步骤见第11.2.5节。

## 11.4 维护日志文件

在启动时，MySQL服务器要检查自己的启动选项以了解是否需要激活日志功能，如果需要，

则打开相应的日志文件。可以让MySQL服务器生成以下几种日志：

- **常规查询日志** 这个日志记载着关于客户连接、数据库查询操作以及其他事件的信息，特别适合用来监控MySQL服务器的运行情况——谁正在建立连接、从何处连接以及他们在做什么。如果想知道客户都向MySQL服务器发来了哪些查询，这个日志是最方便的了；这些信息对故障诊断或调试工作很有价值。
- **慢查询日志** 这个日志能帮你确定哪些SQL语句需要为改善性能而重写。所谓“慢”查询是MySQL服务器根据一个名为long\_query\_time的服务器变量而确定的——如果某个查询花费的时间大于这个变量的值（以秒为单位），MySQL就会认为它是一个慢查询并把它记录到慢查询日志里去。慢查询日志还用于记录没有使用任何索引的查询命令。
- **变更日志** 这个日志记载着对数据库做出了修改的查询命令。这里所说的“修改”并不局限于UPDATE语句，凡是会使数据发生变化的语句都包括在内。因此，这个日志还记载着DELETE、INSERT、REPLACE、CREATE TABLE、DROP TABLE、GRANT和REVOKE等语句的查询。变更日志的内容是一些SQL语句，这些语句可以直接作为mysql程序的输入。以前，这个日志的主要用途是配合各种备份文件去恢复崩溃了的数据库系统（先用备份文件把数据库恢复到当初进行备份时的状态，再把变更日志用做mysql程序的输入以再次执行记载在其中的各种修改操作，最终把数据库恢复到崩溃发生前一刻的状态）。但从MySQL 2.23.14版本开始，随着二进制变更日志的引入和完善，变更日志正逐渐被淘汰。
- **二进制变更日志和二进制日志索引文件** 二进制变更日志与变更日志功用相同，但其内容却是用一种效率更高的二进制格式写出来的，并且还额外增加了一些有用的信息。人们经常把二进制变更日志简称为二进制日志。这个日志主要用来：1）配合备份文件去恢复崩溃了的数据库系统；2）在镜像机制中，把主服务器上的变更情况传输到从服务器。附属于二进制日志的二进制日志索引文件列出了MySQL服务器当前正维护着哪些二进制日志。

在默认情况下，这些日志文件都将被写到MySQL数据目录里，但如果在启动MySQL服务器时没有提出要求，MySQL就不会把它们创建出来。这些日志各自对应着mysqld程序的一个启动选项，只有激活相应的启动选项，MySQL才会把相应的日志文件创建出来。除二进制日志以外，其他日志都是可以直接阅读的ASCII格式。二进制日志的内容要用mysqlbinlog工具程序来查看。

MySQL对错误日志文件的处理方法与其他日志文件不同（比如说，UNIX系统上的错误日志文件是由mysqld\_safe脚本而非MySQL服务器创建的），我们将在第11.4.5节中对它做进一步的讨论。MySQL服务器会为那些特殊的数据表处理程序创建一些专用的日志。ISAM日志主要用于调试工作，它记载着用户对ISAM和MyISAM数据表的修改，在此不对它做详细的介绍。BDB和InnoDB数据表处理程序也有供它们自己内部使用（主要用于崩溃后的自动恢复工作）的日志。

在所有日志中，常规查询日志对MySQL服务器的监控工作是最有用的，所以建议大家在刚开始使用MySQL服务器的时候最好——与你想激活的其他日志一起——激活常规日志。等积累了足够的MySQL使用经验之后，再禁用常规日志以减少磁盘空间的消耗量。

不同的日志要用下面这个表格里不同选项来激活。如果日志文件名可选（见下表中的方括号）而你又没有给出一个文件名，MySQL服务器将使用它的默认名称并把日志文件写到它的



数据目录里去。各个日志文件的默认名是从服务器主机的主机名派生出来的，在以后的讨论里，将用`HOSTNAME`来代表MySQL服务器的主机名。如果给出的日志文件名是一个相对路径名，MySQL服务器将从其数据目录开始对它做出解释；如果给出的是一个绝对路径名，MySQL服务器就将把日志文件放到指定的目录里去。如果日志文件不存在，MySQL服务器将创建之。不过，MySQL服务器不具备创建目录的能力，所以，如果需要创建目录的话，就需要在启动MySQL服务器之前把它们创建出来。

与日志有关的启动选项	由该选项激活的日志
<code>--log[=file_name]</code>	常规日志文件
<code>--log-bin[=file_name]</code>	二进制变更日志文件
<code>--log-bin-index=file_name</code>	二进制变更日志索引文件
<code>--log-update[=file_name]</code>	变更日志文件
<code>--log-slow-queries[=file_name]</code>	慢查询日志文件
<code>--log-isam[=file_name]</code>	ISAM / MyISAM日志文件
<code>--log-long-format</code>	这个选项用来设置慢查询日志和变更日志的格式

如果还激活了BDB或InnoDB数据表处理程序，它们会创建出自己的日志来（在默认情况下，这些日志将被创建在MySQL数据目录里）。虽然不能控制这些日志是否创建，但可以通过以下选项来控制它们将被写到什么地方去：

与BDB或InnoDB日志有关的启动选项	用 途
<code>--bdb-logdir=dir_name</code>	用来设定存放BDB日志文件的目录
<code>--innodb-log_arch_dir=dir_name</code>	用来设定存放InnoDB日志文件的归档目录
<code>--innodb_log_group_home_dir=dir_name</code>	用来设定存放InnoDB日志文件的目录

注意，只要给出了两个InnoDB选项中的任何一个，就必须把另一个也写出来；也就是说，必须同时给出两个InnoDB选项，而且它们的设置值必须相同。

MySQL允许在`mysqld`程序或`mysqld_safe`脚本的命令行上给出各种日志选项。但因为人们在启动MySQL服务器时使用的日志选项几乎总是一样的，所以把它们适当地安排在选项文件中的某个选项组里的做法要更常见一些。日志选项通常都被安排在`[mysqld]`选项组里，但并不是非这样做不可；第11.2.3节对各种MySQL服务器程序和启动脚本所对应的选项组做了详细的介绍。

### 日志的刷新

对日志进行刷新将使MySQL服务器先关闭然后再重新打开日志文件。这个操作可以用`mysqladmin flush-logs`命令或（在MySQL 3.22.9及以后的版本里）`FLUSH LOGS`语句来实现。向MySQL服务器进程发送一个`SIGHUP`信号也将对日志进行刷新。（`mysqladmin refresh`命令也可以对日志进行刷新，但因为这条命令还会做一些其他事情，所以用它去刷新日志有点大材小用。）

日志刷新操作只适用于常规日志、变更日志、二进制变更日志及其索引文件和慢查询日志，但不适用于错误日志和BDB/InnoDB数据表处理程序的专用日志。对于二进制



日志，日志刷新操作将使MySQL服务器关闭它当前使用着的二进制日志文件并按顺序打开带有下一个编号的日志文件——如果变更日志文件也是按某种编号顺序而生成的，日志刷新操作也会导致它发生同样的情况。

日志刷新操作主要用在日志失效或日志轮转工作中，第11.4.6节对此做了详细的讨论。

#### 11.4.1 常规查询日志

常规查询日志（general query log）记载着客户是在何时连接到MySQL服务器的、它们都向MySQL服务器发送了哪些查询以及其他一些与查询无关的事件（如MySQL服务器的启动和关停事件等）。如果用--log选项激活了常规日志却没有给出文件名，它的默认文件名将是MySQL数据目录中的HOSTNAME.log。

查询命令将按它们到达MySQL服务器的先后顺序被记载到这个日志里，这个顺序很可能与它们执行完毕时的顺序不一致，尤其是在长、短查询命令混杂在一起的时候。

#### 11.4.2 慢查询日志

慢查询日志（slow-query log）记载着执行用时较长的查询命令，这里所说的“长”是由MySQL服务器变量long\_query\_time（以秒为单位）定义的。每出现一个慢查询，MySQL服务器就会给它的Slow\_queries状态计数器加上一个1。慢查询日志能帮助你了解哪些查询命令需要改写以改善其执行性能。不过，在对慢查询日志的内容进行分析的时候，应该把系统全局的负载情况也考虑进去：查询命令的执行用时是按人类时间（而非CPU时间）来衡量的，如果MySQL服务器主机出现了拥堵，一些平时执行起来并不慢的查询命令可能会被误判为“慢查询”。

如果用--log-slow-gueries选项激活了慢查询日志却没有给出文件名，它的默认文件名将是MySQL数据目录中的HOSTNAME-slow.log。如果同时使用了--log-slow-queries和--log-long-format选项，MySQL将把那些在执行时没有用到任何索引的查询命令也记载到这个日志里。

因为查询命令的执行用时在它们执行完毕之前无法预测，所以慢查询日志里的查询命令都是在执行完毕后（注意，不是在到达MySQL服务器时）才被记载到这个日志里的。

记载在慢查询日志里的查询命令可以用mysqldumpslow工具程序来查看。

#### 11.4.3 变更日志

变更日志（update log）记载着INSERT、DELETE或UPDATE等会对数据库内容做出修改的查询命令。SELECT语句和下面这样的UPDATE语句——因为它们并没有真正修改什么东西——都不会被记载到变更日志里去：

```
UPDATE t SET i = i;
```

MySQL必须在执行完一条语句之后才能确定它是否修改了数据，所以变更日志里的查询命令是按它们执行完毕的顺序而不是它们到达MySQL服务器的顺序被记载下来的。

在MySQL 3.23.14（二进制变更日志最早出现于这个版本）之前的版本里，变更日志在数据库的备份和恢复工作中发挥着重要作用。但随着既能用在数据库的备份和恢复工作中也能用在镜像机制里的二进制变更日志的引入和完善，变更日志正逐渐被淘汰。

变更日志要用--log-update选项来激活。MySQL服务器将按照以下规则来命名变更日志文件：

- 如果用--log-update选项激活了变更日志却没有给出文件名，MySQL服务器将在其数据目录里以这台服务器的主机名为基本文件名去生成一系列用数字编号的日志文件，即HOSTNAME.001、HOSTNAME.002，依此类推<sup>①</sup>。
- 如果给出了一个不带扩展名的日志文件名，MySQL服务器将用给出的名字代替主机名去作为日志文件的基本文件名并生成一系列用数字编号的日志文件。比如说，如果给出了--log-update=update这样的选项设置，MySQL服务器生成的变更日志文件名将是update.001、update.002，依此类推。
- 如果在激活变更日志时给出了一个带扩展名的日志文件名，MySQL服务器将完全按照所给出的名字去命名变更日志文件而不再生成一系列用数字编号的日志文件。

如果变更日志文件名是用数字编号的，MySQL服务器将在它每次启动或者对日志进行刷新时按顺序生成下一个文件。

如果同时使用了--log-update和--log-long-format选项，MySQL还将把一些附加信息（比如各查询语句都是由哪位用户发出的、是在何时发出的等等）记载到变更日志里去。

#### 11.4.4 二进制变更日志和二进制日志索引文件

类似于变更日志，二进制变更日志（binary update log）也是用来记载对数据库的内容做出了修改的查询命令的，但它的内容却是用一种效率更高的二进制格式而不是ASCII格式写出来的，并且还增加了一些记载信息（如查询命令的时间戳）等。因为其内容是用二进制格式写的，所以二进制变更日志无法直接阅读，但可以用mysqlbinlog工具程序来生成一份可读的二进制日志输出报告。

二进制变更日志既可以用在数据库的备份和恢复工作中，也可以用在镜像机制里。但如果想把某个MySQL服务器用做镜像机制中的主服务器，还必须在启动该服务器时明确地激活有关的选项。

如果用--log-bin选项激活了二进制日志却没有给出文件名，MySQL服务器将以HOSTNAME-bin为基本文件名去生成一系列用数字编号的日志文件，即HOSTNAME-bin.001、HOSTNAME-bin.002，依此类推；如果给出了文件名，MySQL服务器就将以所给出的名字作为基本文件名（如果所给出的文件名带有扩展名，MySQL将去掉那个扩展名）去生成一系列用数字编号的日志文件。每当启动MySQL服务器、对日志进行刷新或者当前日志到达其最大尺寸时，MySQL服务器就会按顺序生成下一个文件。二进制变更日志文件的最大尺寸是由MySQL服务器

<sup>①</sup> 人们正计划把编号形式的日志文件名中的编号从3位数扩大到6位数，这是为了给日志文件名的排序工作提供方便。按照目前的编号方法，当日志文件名中的编号从.999跨至.1000时，对日志文件的排序操作就会混乱。以6位数字做编号就不太容易发生这种排序错乱的情况。

变量`max_binlog_size`的取值来决定的。(注意：二进制变更日志文件的命名规则与变更日志文件的命名规则很相似，但并不完全一样。)

二进制变更日志里的查询命令是按它们的执行顺序被记载下来的。也就是说，MySQL服务器将按有关语句执行完毕的顺序而不是收到这些语句的顺序来记载它们；只有这样，镜像机制才能正确地工作。

组成一个事务的多条语句将被缓存到这个事务被提交之后才会被统一记载到二进制变更日志里去。如果事务被撤销，数据库的内容当然不会发生变化，所以MySQL服务器将不把被撤销的事务记录到二进制变更日志里去。(这与MySQL服务器不把没有实际修改数据库内容的单条查询命令记载到变更日志里去的做法相同；二进制变更日志在遵循同样原则的基础上还将其适用范围扩大到了各有关事务中的多条语句上。)

如果激活了二进制变更日志，MySQL服务器还将同时创建出一个二进制日志索引文件(binary log index file)来，该文件的内容其实就是MySQL服务器当前使用的各二进制变更日志文件的一份清单。二进制日志索引文件的默认文件名与二进制变更日志文件名有着相同的基本文件名，只是扩展名变成了`.index`。可以用`--log-bin-index`选项明确地为二进制日志索引文件另外起一个名字；如果所给出的文件名不带扩展名，MySQL就将自动地把`.index`用做二进制日志索引文件的扩展名。比如说，如果所给出的是`--log-bin-index=binlog`，这个索引文件的名称就将是`binlog.index`。

如果想用二进制变更日志来建立镜像机制，请记住这样一个原则：在把某个二进制变更日志文件的内容施放到镜像机制中所有的从服务器上并确定自己今后肯定不会再用到它之前，千万不要把这个日志文件删掉。第11.4.6节对这方面的问题做了详细的探讨。

### 日志文件与系统备份

如果变更日志或二进制变更日志因为用来存放它们的硬盘出了问题而受到破坏或者丢失，用它们来恢复数据库或者实现镜像机制也就无从谈起了。大家一定要定期对整个文件系统进行备份。把某个数据库与该数据库的日志文件分别存放到不同的硬盘上也是个很不错的办法——这要把各有关文件从它们在MySQL数据目录里的默认存放位置重新安置到别的地方去。重新安置各种日志文件的具体步骤请参见第10章中的有关内容。

### 11.4.5 错误日志

错误日志(error log)记载着MySQL数据库系统的诊断和出错信息。这个日志在UNIX和Windows系统上有着不同的处理方法，对此我们将分别进行讨论。

#### 1. UNIX系统上的错误日志

与其他日志不同，UNIX系统上的错误日志是由负责启动MySQL服务器的`mysqld_safe`脚本而不是由MySQL服务器本身创建的；`mysqld_safe`脚本又是通过对MySQL服务器的标准输出和标准错误输出(即C语言中的`stdout`和`stderr`输出流)进行重定向而创建出错误日志的。错误日志的默认文件名是`HOSTNAME.err`，可以利用`mysqld_safe`脚本的命令行选项`--err-log`或者通过某个

选项文件的[mysqld\_safe]选项组里的err-log语句来给错误日志另外起个名字。(mysqld\_safe脚本在MySQL 4之前的版本里叫做safe\_mysqld。由此前溯到MySQL 3.23.22版本里的safe\_mysqld脚本都支持--err-log选项。再往前, safe\_mysqld脚本就总是用默认文件名来写错误日志了——要想改变这一行为, 只能去直接编辑这个脚本。)

如果在命名错误日志文件时给出的是一个相对路径名, MySQL将从调用mysqld\_safe脚本时所在的目录开始对它做出解释。这与其他日志文件截然不同; 别的日志都由mysqld创建, 而且相对路径名都是从MySQL数据目录开始做出解释的。因为你不见得总是从同一个目录去调用mysqld\_safe脚本(比如说, 可能会根据不同的情况从不同的目录去手动调用mysqld\_safe脚本), 所以大家在命名错误日志文件时最好能给出绝对路径名以保证错误日志总能被创建在同一位置。

还要提醒大家注意的是, 如果错误日志文件已经存在而你用来启动MySQL服务器的登录账户对这个文件没有写权限, MySQL服务器就将无法启动且不会在错误日志里写出任何输出——如果你将使用不同的--user选项值去启动MySQL服务器的话, 就很可能发生这种事情。为避免这一问题, 建议大家最好坚持使用同一个账户去启动MySQL服务器, 对这一问题的详细讨论请参见第11.2.1节。

用mysql.server脚本去启动MySQL服务器也能创建出错误日志来, 这是因为mysql.server脚本会调用到mysql-safe脚本。不过, mysql.server脚本的命令行和它的[mysql\_server]选项组不支持--err-log选项。如果希望在使用mysql.server脚本去启动MySQL服务器时也能指定错误日志的文件名, 就必须在某个选项文件的[mysqld\_safe]选项组里进行设置。

如果是直接执行mysqld程序的办法去启动MySQL服务器的, MySQL会把出错信息输出到终端上而不再去创建错误日志。你自己也可以把MySQL服务器的诊断性输出重定向到其他地方去。比如说, 如果想把出错信息写到一个名为/tmp/mysql.err的文件里去, 那么, 在csh或tcsh里, 就要像下面这样去启动MySQL服务器:

```
% mysqld >& /tmp/mysql.err &
```

或者, 在sh或类似的shell里, 就要像下面这样去启动MySQL服务器:

```
% mysqld > /tmp/mysql.err 2>&1 &
```

## 2. Windows系统上的错误日志

在Windows系统上, MySQL服务器会默认地把诊断信息写到其数据目录中的mysql.err文件里去, 并且不允许另行指定错误日志的文件名。如果在启动MySQL服务器时给出了--console选项, MySQL将把诊断输出到控制台窗口而不再创建错误日志。(如果把MySQL服务器运行为一项服务, -console选项将不起作用, 因为此时不存在什么控制台窗口。)

### 11.4.6 日志文件的失效处理

激活日志功能的弊病之一是因此而产生的大量信息或许会填满磁盘。如果MySQL服务器非常繁忙且需要处理大量的查询, 尤其如此。如果既想保持有足够的日志去记录MySQL服务器的工作情况, 又想防止日志文件无限制地增长, 就需要掌握一些日志文件的失效处理技术。把日志文件控制在可管理范围内的办法主要有以下几种:



- **日志轮转。**这个办法适用于常规查询日志和慢查询日志等文件名固定不变的日志文件。
- **以“年龄”为依据对日志文件进行失效处理。**这个方法的策略是定期删除那些“年龄”大于给定时间的日志文件。它适用于变更日志和二进制变更日志等文件名用数字编号的日志文件。
- **以镜像机制的进展情况为依据对日志文件进行失效处理。**如果要用二进制变更日志文件去建立镜像机制，那就最好不要根据其“年龄”去对日志文件进行失效处理——只有当某个日志文件的内容全都施加到镜像机制中所有的从服务器上之后，才可以考虑把它删除掉。换句话说，是否要删除某个日志文件必须取决于今后是否还要用到它。

对日志进行轮转通常需要有日志刷新操作的配合，这是为了确保缓存在内存里的日志信息都已经被写入磁盘了。可以通过mysqladmin flush-logs命令或FLUSH LOGS语句来完成日志刷新操作。

在本节的后续部分，我们将对日志文件的各种失效技术进行介绍。在选用这些技术的时候，一定要把日志文件与数据库备份策略的相互配合考虑进去。（把数据库恢复工作可能会用到的日志文件也备份下来是一个很好的主意，所以千万别在备份之前就把有用的日志文件删掉了！）以下讨论内容中的示例脚本都可以在sampdb发行版本的admin目录里找到。

#### 1. 对文件名固定不变的日志文件进行失效处理

MySQL服务器会把某些类型的日志信息写到文件名固定不变的日志文件里去。这类日志包括常规查询日志、慢查询日志以及文件名固定不变（即文件扩展名不是数字编号）的变更日志。文件名固定不变的日志文件应该用日志轮转技术来进行失效处理。这种技术允许保留几个最新生成的日志文件，但它们的数量不会超过为防止它们过分消耗磁盘空间而设定的数目。

日志文件的轮转是这样进行的（假设日志文件的名称是log）：在第一次轮转时，把log更名为log.1，然后让MySQL服务器创建一个新的log文件；在第二次轮转时，把log.1更名为log.2，把log更名为log.1，然后让MySQL服务器再创建一个新的log文件；如此循环，每一个日志文件都轮转着被依次更名为log.1、log.2等等；当日志文件轮转到预定的失效位置时，下次轮转就不再对它进行更名了——它将被排在其后面的那个文件覆盖掉。举个例子，如果每天进行一次日志轮转并想保留前后一个星期的日志文件，就需要保留log.1~log.7共七个日志文件；等下次轮转时，用log.6覆盖掉原来的log.7成为新的log.7，原来的log.7自然就失效了。

日志轮转的频率和需要保留的老日志数量取决于MySQL服务器的繁忙程度（服务器越繁忙，生成的日志信息就越多）和你愿意分配多少磁盘空间来存放那些老日志。

UNIX系统允许对MySQL服务器已经打开并正在使用的当前日志文件进行更名，日志刷新操作将关闭当前日志文件并打开一个新日志文件——用原来的名字创建一个新的日志文件。文件名固定不变的日志文件可以用下面这个shell脚本来进行轮转：

```
#! /bin/sh
# rotate_fixed_logs.sh - rotate MySQL log file that has a fixed name

# Argument 1: log file name

if [ $# -ne 1 ]; then
```



```

    echo "Usage: $0 logname" 1>&2
    exit 1
fi

```

```
logfile=$1
```

```

mv $logfile.6 $logfile.7
mv $logfile.5 $logfile.6
mv $logfile.4 $logfile.5
mv $logfile.3 $logfile.4
mv $logfile.2 $logfile.3
mv $logfile.1 $logfile.2
mv $logfile $logfile.1
mysqladmin flush-logs

```

这个脚本以日志文件名作为参数，既可以直接给出日志文件的完整路径名，也可以先进入日志文件所在的目录再给出日志文件的文件名。比如说，如果想对/usr/mysql/data目录名为log的日志进行轮转，可以使用下面这条命令：

```
% rotate_fixed_logs.sh /usr/mysql/data/log
```

也可以使用下面的命令：

```

% cd /usr/mysql/data
% rotate_fixed_logs.sh log

```

为确保自己总是有权对日志文件进行更名，最好是在以mysqladm为登录名上机时才运行这个脚本。还要提醒大家注意的是，在这个脚本里的mysqladmin命令行上没有给出-u或-p之类的连接选项参数——如果你已经把执行mysql客户程序时要用到的连接参数保存到了mysqladm程序的.my.cnf选项文件里，就用不着在这个脚本中的mysqladmin命令行上再次给出它们了；如果你没有使用选项文件，就得用-u和-p选项告诉mysqladmin使用哪个MySQL账户（这个MySQL账户必须具备日志刷新操作所需要的权限）去连接MySQL服务器。不过，因为那个MySQL账户的口令要出现在rotate\_fixed\_logs.sh脚本的代码里，所以，为防止这个脚本成为安全漏洞，建议大家专门创建一个除了能对日志进行刷新以外其他什么事都不能做的MySQL账户（即一个具备且仅具备RELOAD权限的MySQL账户），把该账户的口令写到脚本代码里，最后再把这个脚本设置成只允许mysqladm用户去编辑和使用。下面这条GRANT语句将以“flush”为用户名、以“flushpass”为口令创建一个如上所述的MySQL账户来：

```
GRANT RELOAD ON *.* TO 'flush'@'localhost' IDENTIFIED BY 'flushpass';
```

创建出这个账户之后，再把rotate\_fixed\_logs.sh脚本中的mysqladmin命令行改写为如下所示的样子：

```
mysqladmin -u flush -pflushpass flush-logs
```

要想定期进行日志轮转和刷新，请参见后面“让日志失效处理工作自动进行”小节内容。

Linux系统上的MySQL发行版本带有一个用来安装mysql-log-rotate日志轮转脚本的logrotate工具，所以不必非得使用rotate\_fixed\_logs.sh或者自行编写其他的类似脚本。mysql-log-rotate脚

本在RPM发行版本里的存放位置是/usr/share/mysql目录，在二进制发行版本里的存放位置是MySQL安装路径下的support-files目录，在MySQL源代码发行版本里的存放位置是share/mysql目录。

Windows系统上的日志轮转与UNIX系统的不太一样。如果你试图对一个已经被MySQL服务器打开并使用着的日志文件进行更名操作，就会发生“file in use”（文件已被打开）错误。要在Windows系统上对日志进行轮转，就得先关停MySQL服务器，然后对文件进行更名，最后再重新启动MySQL服务器，在Windows系统上启动和关停MySQL服务器的步骤前面已经介绍过了，下面是用来对日志文件进行更名的批处理文件：

```
@echo off
REM rotate_fixed_logs.bat - rotate MySQL log file that has a fixed name

if not "%1" == "" goto ROTATE
    @echo Usage: rotate_fixed_logs logname
    goto DONE

:ROTATE
set logfile=%1
erase %logfile%.7
rename %logfile%.6 %logfile%.7
rename %logfile%.5 %logfile%.6
rename %logfile%.4 %logfile%.5
rename %logfile%.3 %logfile%.4
rename %logfile%.2 %logfile%.3
rename %logfile%.1 %logfile%.2
rename %logfile% %logfile%.1
:DONE
```

这个批处理程序的用法与rotate\_fixed\_logs.sh脚本差不多，它也需要你提供一个将被轮转的日志文件名作为参数。如下所示：

```
C:\> rotate_fixed_logs C:\mysql\data\log
```

或者如下所示：

```
C:\> cd \mysql\data
C:\> rotate_fixed_logs log
```

在最初几次执行日志轮转脚本的时候，日志文件的数量尚未达到预设的上限值，脚本会报告说找不到某几个文件；这是正常的。

## 2. 对文件扩展名以数字编号的日志文件进行失效处理

文件名固定不变的日志文件可以用轮转文件名的办法来进行失效处理，这前面刚讨论过。对变更日志和二进制变更日志等以数字编号作为扩展名的日志文件来说，日志的失效处理工作要用稍微不同的办法来完成。此时，日志文件应该根据其“年龄”（这可以利用文件的最后一次修改时间计算出来）而不是使用一组给定的文件名进行轮转来对它们进行失效处理。这么做的理由是：前、后两个编号型日志的创建时间不见得都有同样的时间间隔，所以不能想当然地认

为保留最新的 $n$ 个日志就会万无一失。如果MySQL服务器在短时间内连续进行了好几次日志刷新操作，它就会创建出好几个日志文件，但它们却不一定都“老”到了可以丢弃的地步。

下面是一个用来对以数字编号作为扩展名的日志文件进行失效处理的脚本：

```
#!/usr/bin/perl -w
# expire_numbered_logs.pl - Look through a set of numbered MySQL
# log files and delete those that are more than a week old.

# Usage: expire_numbered_logs.pl logfile ...

use strict;
die "Usage: $0 logfile ...\n" if @ARGV == 0;
my $max_allowed_age = 7;      # max allowed age in days
foreach my $file (@ARGV)      # check each argument
{
    unlink ($file) if -e $file && -M $file >= $max_allowed_age;
}
exit (0);
```

这个脚本是用Perl语言写的。Perl一种跨平台的脚本语言，用它编写出来的脚本在UNIX和Windows系统上都能使用。这个脚本也需要你提供一个将被轮转的日志文件名作为参数，下面是它在UNIX系统上的用法：

```
% expire_numbered_logs.pl /usr/mysql/data/update.[0-9]*
```

或者像下面这样：

```
% cd /usr/mysql/data
% expire_numbered_logs.pl update.[0-9]*
```

注意，如果传递给这个脚本的文件名参数不正确，它就会非常危险！比如说，如果出现这样的失误（把“\*”作为这个脚本的文件名参数）：

```
% cd /usr/mysql/data
% expire_numbered_logs.pl *
```

它就会把/usr/mysql/data目录里“年龄”大于7天的所有文件——不仅仅是日志文件——全都删掉。

### 3. 对与镜像机制有关的日志文件进行失效处理

对于MySQL服务器生成的数字编号型二进制变更日志，可以像刚才介绍的那样按“年龄”来进行失效处理。但如果还把二进制日志用在镜像机制中，按“年龄”来判断它是否应该被删除就不合适了。此时，日志文件只有在其内容被施加到所有的从服务器上之后才可以删除。

但是，镜像机制中的主服务器本身既不知道自己有多少个从MySQL服务器，也不知道哪个日志文件已经被施加到所有的从服务器上了。要知道，虽说主服务器不会“自做主张”地把尚未发送给从服务器的二进制日志删掉，但很难保证某给定从服务器在任意给定时刻都会与主服务器保持着连接。这意味着：1）你本人必须知道主服务器到底有几个从服务器、哪些从服务器正在运行着；2）必须由你依次连接每一个从服务器并发出SHOW SLAVE STATUS语句才能确

定它正处理着主服务器的哪一个二进制日志文件（这个日志文件的文件名将出现在SHOW SLAVE STATUS语句所输出的Master\_Log\_File列里）。只有所有的从服务器都不再会用到的二进制日志才可以被删掉。我们来看一个例子，假设：

- 本地MySQL服务器是主服务器，它有两个从MySQL服务器S1和S2。
- 在主服务器上有5个二进制日志文件，它们的名字是binlog.038~binlog.042。
- SHOW SLAVE STATUS语句在S1上的执行结果是：

```
mysql> SHOW SLAVE STATUS\G
...
Master_Log_File: binlog.41
...
```

在S2上的执行结果是：

```
mysql> SHOW SLAVE STATUS\G
...
Master_Log_File: binlog.40
...
```

这样，我们就知道从服务器仍在使用的、最低编号的二进制日志是binlog.40，而编号比它更小的那些二进制日志——因为不再有从服务器需要用到它们——已经可以安全地删掉了。于是，连接到主服务器并发出下面的语句：

```
mysql> PURGE MASTER LOGS TO 'binlog.040';
```

在主服务器上发出的这条命令将把编号小于40的二进制日志文件——就这个例子来说，就是把binlog.38和binlog.039全都删掉。

SHOW SLAVE STATUS语句最早出现于MySQL 3.23.22版本，PURGE MASTER LOGS语句最早出现于MySQL 3.23.28版本。要想执行这两条语句，必须拥有SUPER权限（在MySQL 4.0.2之前的版本里，必须拥有PROCESS权限）。

#### 4. 让日志失效处理工作自动进行

以手动方式来执行日志失效处理脚本没什么不好，可如果能安排有关命令自动执行的话，它们就不必由你来定期地执行了。让日志失效处理工作自动进行的办法之一是用cron工具程序创建一个crontab文件来定期删除不再有用的日志文件。不熟悉cron的读者可以用下面的命令去查一下自己的UNIX系统手册：

```
% man cron
% man crontab
```

crontab文件的格式可能要用下面这条命令才能查出来：

```
% man 5 crontab
```

现在，假设：1）用rotate\_fixed\_logs.sh脚本来轮转常规日志、用expire\_numbered\_logs.pl脚本来对编号型变更日志进行失效处理；2）这两个脚本都安装在/u/mysqladm/bin目录里。先以mysqladm账户登录上机，然后用下面这条命令编辑mysqladm用户的crontab文件：

```
% crontab -e
```

这条命令将打开当前crontab文件（如果此前未设置过cron作业，这个文件可能是空的）的一个副本供你进行编辑。把下面两行添加到crontab文件里：

```
0 4 * * * /u/mysqladm/bin/rotate_fixed_logs.sh /usr/mysql/data/log
0 4 * * * /u/mysqladm/bin/expire_numbered_logs.pl /usr/mysql/data/update.[0-9]*
```

这些设置项告诉cron要在每天凌晨4点去执行那两个脚本。可以根据自己的具体情况修改这个时间；crontab设置项的具体格式可以在crontab手册页中查到。如果MySQL服务器比较繁忙，请适当提高日志失效处理工作的频率，因为繁忙的MySQL服务器会比不那么繁忙的MySQL服务器生成更多的日志信息。

如果还想定期地对日志进行刷新（比如为了生成下一编号的变更日志或二进制变更日志），可以再增加一个crontab设置项去定期执行mysqladmin flush\_logs命令。为保证cron总能找到mysqladmin，应该在crontab文件里写出mysqladmin程序的完整路径名。

## 11.5 其他MySQL服务器配置问题

这一节将介绍几种对MySQL服务器进行定制或有助于改善其性能的配置方法，主要包括：

- 对MySQL服务器如何在网络接口上监听客户连接的情况进行控制。
- 激活或者禁用LOAD DATA语句的LOCAL能力。
- 国际化和本地化问题，比如设置MySQL服务器的地理时区和字符集等。
- 激活或者禁用BDB/InnoDB数据表处理程序。
- 配置InnoDB数据表处理程序。
- 通过设置内部变量来优化MySQL服务器。

### 11.5.1 对MySQL服务器的连接监听情况进行控制

可以对MySQL服务器在以下几种网络接口上的连接监听情况进行控制：

- 任何一种平台上的MySQL服务器都监听着供TCP/IP连接使用的网络接口——除非在启动它的时候给出了--skip-networking选项。MySQL服务器使用的默认端口号是3306；可以用--port选项另行指定一个不同的端口号。如果MySQL服务器主机有多个IP地址，还可以用--bind-address选项对MySQL服务器在监听客户连接时使用的IP地址进行设定。
- 在UNIX系统下，MySQL服务器还将在一个UNIX域套接字文件上监听有无本地客户在试图以localhost为主机名进行连接。默认的套接字文件是/tmp/mysql.sock，可以用--socket选项另行指定一个不同的文件名。
- 在基于NT的Windows系统上，名字里有-nt字样的MySQL服务器都支持命名管道。默认的命名管道名是MySql；可以用--socket选项另行指定一个不同的命名管道名。在MySQL 3.23.50之前的版本里，对命名管道的支持总是激活的；在MySQL 3.23.50及以后的版本里，对命名管道的支持是默认禁用的，必须用--enable-named-pipe选项明确地激活它才能使用命名管道。

如果只运行了一个MySQL服务器，让它在默认的网络接口上进行监听是比较典型的做法。



如果运行有多个MySQL服务器，就必须保证不同的MySQL服务器有它们各自专用的网络参数。我们将在第11.6节对此做进一步讨论。

以上介绍只适用于运行在客户/服务器环境里的独立式MySQL服务器。被链接到应用程序里的嵌入式MySQL服务器与其客户端之间的通信是通过内部链路实现的，根本不需要去监听任何外部的网络接口，所以这里的讨论不适用于嵌入式MySQL服务器。

### 11.5.2 激活或者禁用LOAD DATA语句的LOCAL能力

在MySQL 3.23.49及以后的版本里，在编译或启动MySQL服务器时都可以激活或者禁用LOAD DATA语句的LOCAL能力：

- 在MySQL服务器的编译阶段，可以在运行configure脚本时用--enable-local-infile或--disable-local-infile选项把LOAD DATA语句的LOCAL能力设置为默认激活或默认禁用状态。
- 在MySQL服务器的启动阶段，可以用--local-infile或--disable-local-infile选项来激活或者禁用MySQL服务器端的LOCAL能力。（在MySQL 4.0.2之前的版本里，要用--local-infile=0来禁用之。）

如果在MySQL服务器端禁用了LOCAL能力（没有把它编译到MySQL服务器里或者在启动MySQL服务器时禁用了它），客户端将不能使用这一能力。即使在MySQL服务器端激活了LOCAL能力（把它编译到了MySQL服务器里并在启动MySQL服务器时激活了它），MySQL客户程序开发库仍会把客户端的LOCAL能力设置为默认禁用状态，但某些客户程序能够有选择地激活之。比如说，mysql程序的--local-infile选项就是用来激活客户端LOCAL能力的。

### 11.5.3 国际化和本地化问题

这里所说的“国际化”指的是软件能够在世界多个地区使用，“本地化”则指的是从软件的国际化支持中选择一套适用于本地区的支持来使用。与国际化和本地化有关的MySQL配置问题主要有以下几个方面：

- MySQL服务器的地理时区。
- MySQL服务器将使用何种语言来显示诊断信息和出错信息。
- 有哪些字符集可供MySQL服务器选用以及它具体选用的默认字符集。

#### 1. 设置MySQL服务器的地理时区

如果MySQL服务器的地理时区设置得不正确，它报告出来的当地时间（比如北京时间）也就不正确，这需要由你来明确地加以纠正。需要注意的是，在UNIX系统上为MySQL服务器设置地理时区必须通过负责启动MySQL服务器的safe\_mysqld或mysqld\_safe脚本而不是通过服务器程序mysqld本身来进行。

地理时区要用--timezone选项来设置。建议把这个选项放到某个选项文件里——尤其是在MySQL服务器是通过不支持命令行选项的mysql.server脚本调用safe\_mysqld或mysqld\_safe脚本来启动的时候。比如说，可以把用来设定美国中部时间的指令添加到选项文件的[mysqld\_safe]选项组里：

```
[mysqld_safe]
timezone=US/Central
```

上例中的语法适用于包括Solaris、Linux或Mac OS X在内的绝大多数UNIX系统。另一种常见的语法是：

```
[mysqld_safe]
timezone=CST6CDT
```

请大家根据自己系统的具体情况去选用相应的语法。

mysqld\_safe脚本在MySQL 4之前的版本里叫做safe\_mysqld，由此一直前溯到3.23.28版本的mysqld\_safe脚本还支持命令行选项--timezone——它的命令行语法与mysqld\_safe脚本所使用的完全一样，只是在选项文件里要被放到[safe\_mysqld]选项组里去。MySQL 3.23.28版本之前的mysqld\_safe脚本不支持--timezone选项，所以只能像下面这样修改safe\_mysqld脚本本身（这些用来设置环境变量TZ的指令必须添加在那条用来启动MySQL服务器程序的指令前面）：

```
TZ=U.S./Central
export TZ
```

或者像下面这样：

```
TZ=CST6CDT
export TZ
```

## 2. 选择用来显示出错信息的语言

MySQL服务器能够用好几种语言来显示诊断信息与出错信息。它的默认值是english（英语），但可以另行设置其他语言。如果想知道都有哪些语言可供选用，请查看MySQL安装路径下的share/mysql目录里有几个以语言名称作为名字的下级目录。如果想改用另一种语言，请用--language启动选项给出该种语言的名称或路径名。以法语为例，如果把MySQL安装在了目录/usr/local/mysql下，就要用--language=french或--language=/usr/local/mysql/share/mysql/french来改变出错信息的显示语言。

## 3. 配置MySQL服务器支持的字符集

MySQL支持多种字符集。字符集的选用不仅会影响在字符串里都能使用哪些字符，还会影响到基于字符串的排序操作以及数据表和数据列的合法命名字符。本小节的讨论重点是如何配置MySQL服务器的字符集支持，它们对实际工作的影响和具体用法见第2章。

如果你想知道自己的MySQL服务器都配备有哪几种可供选用的字符集，可以查看MySQL安装路径下的share/mysql/charsets目录，其中的Index文件列出了一份可用的字符集清单。下面这个查询也可以把这份清单查出来：

```
mysql> SHOW VARIABLES LIKE 'character_sets';
```

或者，在MySQL 4.1及以后的版本里，用SHOW CHARACTER SET语句也能查出这份清单和一些相关信息。

如果想从可用的字符集中挑出一个作为MySQL服务器的默认字符集，就要在MySQL服务器的编译阶段通过configure脚本的下列选项来进行配置：

- MySQL服务器的默认字符集是latin1，但可以用--with-charset选项另行指定一个。
- 如果想给MySQL服务器增加对其他字符集的支持，就要使用--with-extra-charsets选项。这个选项的参数是一个以逗号为分隔符的字符集清单。比如说，如果想让MySQL服务器支持latin1、big5和hebrew字符集，就要像下面这样去配置MySQL软件的源代码发行版本：

```
% ./configure --with-extra-charsets=latin1,big5,hebrew
```

--with-extra-charset选项有两个特殊的参数值：all代表所有的可用字符集；complex代表所有的复杂字符集。所谓“复杂字符集”包括多字节字符集和有特殊排序规则的字符集。

在启动时，MySQL服务器将默认地在编译阶段用--with\_charset选项指定的字符集作为它的默认字符集。如果想用另外一个作为默认字符集，就要在启动MySQL服务器时用--default-character-set选项另行设置。

虽说MySQL服务器可以使用不同的字符集，但MySQL 4.1之前的版本并不支持同时使用多个字符集。在MySQL 4.1及以后的版本里，用来添加可用字符集和指定默认字符集的编译配置选项和启动选项仍与以前的版本一样，但增加了在服务器、数据库、数据表、数据列和字符串常数级临时改用另一种字符集的功能（即所谓“SQL级支持”）。换句话说，MySQL服务器现在支持同时使用多个字符集了。随着字符集支持能力的提高，人们同时使用的字符集越来越多，这就要求你在编译阶段把更多的可用字符集纳入MySQL服务器。（比如说，有很多用户都盼望着能早日用上Unicode字符集，所以在编译MySQL服务器的时候就应该把它纳入进去。）

在MySQL 4.1版本之前，如果在创建好数据表之后又改变了MySQL服务器的默认字符集，就可能需要对索引重新排序才能保证索引键值能够正确地反映出数据表记录在新字符集下的排列顺序。对于MyISM数据表，既可以同时使用myisamchk程序的--recover和--quick选项——再加上负责指定新字符集的--set-character-set选项——来重新对索引进行排序（注意，在执行myisamchk程序之前先要关停MySQL服务器），也可以在不关停MySQL服务器的前提下用REPAIR TABLE ... QUICK语句或mysqlcheck --repair --quick命令来重新对索引进行排序。第三种办法是按“转储 - 丢弃 - 重新加载”的顺序对各有关数据表进行处理，这个办法适用于包括MyISAM数据表在内的各种数据表。在MySQL 4.1及以后的版本里，改善了字符集支持能力使人们在改变了默认字符集之后不必再去重新建立索引了。不过，需要把老数据表的格式升级到4.1版本才能享受到这个好处，其具体步骤见下一小节。

在客户端，可以用--default-character-set选项告诉客户程序使用哪个字符集作为它的默认字符集。如果没有把新字符集安装到它们的默认位置（MySQL安装路径下的share/mysqlCharsets目录），而是把所需的字符集文件安装到了另一个目录下，还可以用--character-sets-dir选项把这个地点告知客户程序。

#### 4. 对老数据表进行转换以激活MySQL 4.1的字符集支持

为了能充分地享受到改善了字符集支持能力所带来的好处，在把老版本的MySQL升级到4.1或更高版本的时候，建议大家最好把数据表也同时升级到4.1版的格式。下面先介绍第一种做法：

##### 1) 用mysqldump程序对数据库进行备份：

```
% mysqldump -p -u root --all-databases --opt > dumpfile.sql
```

--all-databases选项的作用是转储所有的数据库；--opt选项的作用是对转储文件进行优化，优化后的转储文件在重新加载时会处理得更快一些。（mysqldump程序将在第13章介绍。）

2) 关停MySQL服务器。

3) 对MySQL软件进行升级并重新启动MySQL服务器，但现在先不改变MySQL服务器的默认字符集。

4) 用备份文件重新加载数据表（数据表将自动转换为4.1格式）：

```
% mysql -p -u root < dumpfile.sql
```

在此过程中，MySQL服务器将把新的字符集支持信息安装到数据表里，这主要体现在以下两个方面：

- 把MySQL服务器的字符集信息分配给每一个数据列。此后，即便改变了MySQL服务器的默认字符集，各数据列也将继续使用自己原来的字符集而不会受到影响。（因为发生在服务器级的字符集改变对数据列完全没有影响，数据列上的索引也就不再会行为失常了。）
- 当以后修改某个数据列的字符集时，MySQL服务器将自动地对与该数据列有关的各个索引重新进行排序，使它们能够正确地反映出数据记录在新字符集下的排列顺序。

下面再来看看第二种做法（不推荐这种做法，它既麻烦又容易出错）：先升级MySQL服务器，再用ALTER TABLE语句来转换数据表的格式。假设有一个结构定义如下的数据表：

```
CREATE TABLE t
(
    c1 CHAR(10),
    c2 CHAR(10),
    c3 CHAR(10)
);
```

为了给各数据列添加上字符集信息，需要明确地用下列语句对之进行转换：

```
ALTER TABLE t
    MODIFY c1 CHAR(10) CHARACTER SET latin1,
    MODIFY c2 CHAR(10) CHARACTER SET latin1,
    MODIFY c3 CHAR(10) CHARACTER SET latin1;
```

这实在是太麻烦了——需要对每一个数据表的每一个数据列进行转换。按“先转储，再重新加载”顺序进行的第一种做法无疑要简单得多。

#### 11.5.4 选择数据表处理程序

MySQL支持多种数据表处理程序。它们有些内建在MySQL服务器程序里，有些则允许在编译阶段省略；在那些内建的数据表处理程序当中又有一些允许在启动MySQL服务器时禁用：

- 在MySQL 4之前的版本里，ISAM处理程序总是内建的。在MySQL 4及以后的版本里，可以在运行MySQL服务器程序的编译配置脚本configure时用--without-isam选项省略掉ISAM处理程序。

对于嵌入式MySQL服务器，ISAM处理程序是默认省略的——如果想把它添加到MySQL服务器程序里，就必须对源发行版本中的mysql\_embed.h文件进行编辑并重新编译MySQL



服务器。但建议大家把ISAM数据表都转换为MyISAM数据表并不再继续使用ISAM存储格式。ISAM支持很可能会在不远的将来彻底退出历史舞台。

- BDB处理程序可以用编译配置脚本configure的--with-berkeley-db选项内建到MySQL服务器程序里。已经内建在MySQL服务器程序里的BDB处理程序可以在启动该服务器时用--skip-bdb选项禁用掉。
- 在MySQL 4版本之前，InnoDB处理程序需要在运行MySQL服务器程序的编译配置脚本configure时明确地给出--with-innodb选项才会被内建到MySQL服务器程序里；在MySQL 4及以后的版本里，InnoDB是默认内建的，但可以用configure脚本的--with-innodb选项把它排除在MySQL服务器程序外。已经内建在MySQL服务器程序里的InnoDB处理程序可以在启动服务器时用--skip-innodb选项禁用掉。
- 从MySQL 3.23版本开始，MyISAM处理程序总是内建的；既不能在编译阶段省略之，也不能在启动MySQL服务器时禁用之。

### 11.5.5 配置InnoDB表空间

InnoDB数据表处理程序不像其他的数据表处理程序那样会为每个数据表分别创建一些文件，它会把所有的InnoDB数据表都存放在同一个表空间里并把整个表空间当做一个在逻辑上连续不断的存储块来对待，与各InnoDB数据表相关联的独立文件只有它们的.frm定义文件（这些.frm定义文件分别存放在各InnoDB数据表所属的数据库的数据库目录里。从某种意义上讲，InnoDB表空间就像是一个虚拟文件系统。）

InnoDB表空间在逻辑上被视为一个连续不断的存储区域，但实际上却是由一个或者多个磁盘文件组成的。各组成文件既可以是一个普通的文件，也可以是一个未格式化的原始硬盘分区。在这一小节里，我们将对InnoDB表空间的配置和管理选项进行介绍。可以在MySQL服务器的启动命令行上设定这些选项，但实际上这种做法很少见。为保证自己在每次启动MySQL服务器时使用的都是同一套InnoDB配置，应该把InnoDB表空间的配置和启动选项放到某个选项文件里的某个适当的服务器选项组（比如[mysqld]或[server]选项组）里去。在InnoDB表空间的配置和启动选项里，下面两个是最重要的：

- innodb\_data\_home\_dir：用来设定InnoDB表空间各组成文件的父目录（人们把这个目录称为“InnoDB主目录”）。如果没有给出这个选项，它的默认值将是相应的MySQL数据目录。
- innodb\_data\_file\_path：对InnoDB主目录中的各有关文件的描述。这个选项的值是由InnoDB表空间各组成文件的规格说明构成的一个列表，各文件的规格说明以分号间隔。每个文件的规格说明由文件名、文件长度和一些可能出现的选项组成，它们彼此以冒号间隔。InnoDB表空间各组成文件的总长度至少为10MB。

在MySQL 3.23版本里，必须给出一个innodb\_data\_file\_path选项值，否则InnoDB处理程序就无法正确启动（其后果之一是MySQL服务器也无法启动。可以通过错误日志去了解MySQL服务器启动失败的原因是否与InnoDB有关）。在MySQL 4版本里，如果没有给出一个innodb\_data\_file\_path选项值，MySQL服务器将自动创建一个仅由一个名为ibdata1的单个文件构成的默认InnoDB表空间（在MySQL 4.0.0和4.0.1版本里，这个默认InnoDB表空间是一个长度



为64MB的非自动扩展文件；在以后的版本里，它是一个长度为10MB的自动扩展文件）。

举一个简单的例子，假设你在MySQL数据目录里创建了一个由两个长度都是10MB的innodata1和innodata2文件组成的表空间。下面就是这个表空间的配置情况：

```
innodb_data_file_path = innodata1:10M;innodata2:10M
```

此时，用不着给出一个innodb\_data\_home\_dir选项值，因为innodata1和innodata2文件就存放在该选项的默认值——这个MySQL服务器的数据目录——所指示的目录里。

InnoDB处理程序将按以下规则去组合innodb\_data\_home\_dir和innodb\_data\_file\_path选项的设置值以确定表空间文件的路径名：

- 如果innodb\_data\_home\_dir选项值为空，innodb\_data\_file\_path选项所给出的InnoDB表空间各组成文件的规格说明中的文件路径名将被解释为绝对路径名。
- 如果innodb\_data\_home\_dir选项值不为空，它就是一个目录名，而innodb\_data\_file\_path选项所给出的InnoDB表空间各组成文件的规格说明中的文件路径名都将被解释为这个目录下的相对路径名。
- 如果innodb\_data\_home\_dir选项根本就没有给出，它的默认值将是MySQL数据目录的路径名，而innodb\_data\_file\_path选项所给出的InnoDB表空间各组成文件的规格说明中的文件路径名都将被解释为该数据目录下的相对路径名。

根据上述规则，如果MySQL数据目录是/var/mysql/data，以下三组配置所指定的文件就将是完全相同的：

```
innodb_data_home_dir=
innodb_data_file_path=/var/mysql/data/ibdata1:10M;/var/mysql/data/ibdata2:10M

innodb_data_home_dir=/var/mysql/data
innodb_data_file_path=ibdata1:10M;ibdata2:10M

innodb_data_file_path=ibdata1:10M;ibdata2:10M
```

在innodb\_data\_file\_path选项值里，各有关文件的规格说明要用分号隔开，规格说明中的各个部分要用冒号隔开。最简单的规格说明语法由一个文件名和一个文件长度值组成，但下列语法也是合法的：

```
path:size
path:size:autoextend
path:size:autoextend:max:maxsize
```

第一种格式给出的是一个长度固定为size的文件。size必须是一个正整数，它后面还要有一个代表兆字节或吉字节的字母“M”或“G”。第二种格式给出的是一个可自动扩展文件；当文件被填满时，InnoDB将以8MB为步长扩展之。第三种格式与第二种类似，但增加了一个用来规定该文件最大长度的数字。MySQL 3.23.50及以后的版本都允许使用可自动扩展的表空间组成文件，但只允许表空间的最后一个组成文件自动扩展。

要想创建InnoDB表空间，只需把有关指令添加到选项文件里（要保证表空间组成文件都不存在）再启动MySQL服务器就行了；InnoDB将注意到有关文件尚未存在并把它们创建和初始化

出来。

MySQL 3.23.41 及以后的版本还允许把未格式化的原始硬盘分区用做InnoDB表空间的组成部分。这样做的好处之一是能轻易创建一个非常巨大的InnoDB表空间：首先，整个原始硬盘分区都可以用来存放数据，而普通文件的最大长度却要受到操作系统的限制；其次，原始硬盘分区能保证整个存储空间的连续性，而普通文件却会受到文件系统碎片化的影响。为减少普通文件的碎片化问题，在对表空间进行初始化的时候，InnoDB会尽量采用向有关文件写入足够多的零的办法去迫使操作系统把存储空间一次性地全部分配给它们而不是以递增方式去分配存储空间。这有助于减少碎片，但不能从根本上杜绝之。

用原始硬盘分区来构成需要两个步骤。假设打算用路径名/dev/rds8:2Gnewraw把一个2GB的原始硬盘分区分配给InnoDB表空间。此时，因为这个原始硬盘分区位于MySQL数据目录以外，所以首先要设定一个innodb\_data\_home\_dir选项值。下面是具体的配置步骤：

1) 在这个原始硬盘分区的长度值后面加上一个newraw后缀以表明这个InnoDB表空间组成文件其实是一个未格式化的硬盘分区，需要InnoDB对它进行初始化：

```
innodb_data_home_dir =
innodb_data_file_path = /dev/rds8:2Gnewraw
```

把这两行添加到[mysqld]选项组并启动MySQL服务器，InnoDB将看到后缀并对这个分区进行初始化。（InnoDB会以只读方式处理这个表空间，因为它知道你还没有完成第二个步骤。）完成了硬盘分区的初始化工作之后，关停MySQL服务器。

2) 修改配置信息，把后缀newraw改为raw：

```
innodb_data_home_dir =
innodb_data_file_path = /dev/rds8:2Graw
```

然后再次启动MySQL服务器。因为后缀中不再有new字样，InnoDB将明白这个硬盘分区已进行过初始化，它就会以读/写方式去使用这个表空间了。

把某个原始硬盘分区用做InnoDB表空间的一部分还要注意以下几个细节：首先，必须把这个硬盘分区的访问权限设置成允许MySQL服务器去读/写；其次，必须保证这个硬盘分区没被用于其他目的——如果有两个或更多个进程往这个分区上写数据，就会把事情弄得一团糟。比如说，如果错误地把系统数据交换分区用做了InnoDB表空间，系统肯定要崩溃！

当在Windows系统上配置InnoDB表空间的时候，Windows路径名中的反斜线字符可以写成单个的斜线字符（“/”），也可以写成两个反斜线字符（“\\”）。此外，尽管在文件路径名里可能会出现冒号（完整的Windows路径名是以一个硬盘盘符和一个冒号开头的），但仍必须使用冒号来分隔各有关文件的规格说明。在遇到冒号的时候，InnoDB会查看它后面的字符并消除其二义性：如果后面的字符是一个数字，就表明是某InnoDB组成文件的长度；否则，就说明仍是一个路径名。请看下面这个配置，它创建的InnoDB表空间将由位于C盘和D盘上的两个长度分别为50MB和60MB的文件组成：

```
innodb_data_home_dir =
innodb_data_file_path = C:/ibdata1:50M;D:/ibdata2:60M
```

如有必要, InnoDB会在每次启动时把尚不存在的表空间数据文件创建出来。它还会创建一些日志文件(如果它们尚不存在的话)。在默认情况下, InnoDB日志文件都将创建在MySQL服务器的数据目录里并以“ib\_”作为文件名的前几个字符。但要提醒大家注意的是, InnoDB只能创建文件, 不能创建目录, 所以必须在启动MySQL服务器之前把InnoDB要用到的目录都创建好(具体做法请参见第11.4节)。

在首次创建InnoDB表空间的时候, 如果MySQL服务器因为InnoDB没能创建出某些必要的文件而启动失败, 请查阅MySQL服务器的错误日志以分析其原因。找出问题的根源之后, 先删除InnoDB已经创建出来的所有文件(用来构成InnoDB表空间的原始硬盘分区不包括在内), 再改正配置错误, 最后重新启动MySQL服务器。

一旦InnoDB完成了对表空间的初始化工作, 就不能再改变InnoDB表空间各组成文件的长度了。不过, MySQL允许在现有的InnoDB表空间组成文件清单里增加一个新文件, 这在InnoDB表空间快被数据填满时非常有用。InnoDB表空间快被填满的征兆之一是一些本应该成功的InnoDB事务操作无法完成。还可以用下面这条语句去了解还剩下多少可用空间, 语句中的`tbl_name`可以是任何InnoDB数据表的名字:

```
mysql> SHOW TABLE STATUS LIKE 'tbl_name';
```

如果想通过增加一个新组成文件的办法来扩大InnoDB表空间, 请按以下步骤进行:

1) 关停正在运行的MySQL服务器。

2) 如果InnoDB表空间的最后一个组成文件是可自动扩展的, 就必须先把它改变为一个固定长度文件才能把另一个文件追加到它的后面。先查出这个文件的实际长度并把它换算成最接近的MB数(要按 $1\text{MB} = 1\,048\,576$ 个字节计算, 不要按 $1\text{MB} = 1\,000\,000$ 个字节计算), 再把计算结果写到其规格说明里。比如说, 假设InnoDB表空间由一个名为`ibdata1`的文件构成:

```
innodb_data_file_path = ibdata1:100M:autoextend
```

如果这个文件现在的实际长度是 $121\,634\,816$ 个字节, 长度换算的结果就将是 $116\text{MB}$  ( $121\,634\,816 \div 1\,048\,576 = 116$ )。所以要把它的规格说明修改为:

```
innodb_data_file_path = ibdata1:116M
```

3) 把新的InnoDB表空间组成文件追加在现有文件清单的末尾。如果新组成文件是一个普通文件, 必须保证它此时尚不存在; 如果新组成文件是一个原始硬盘分区, 请按前面刚介绍过的两个步骤把它添加到InnoDB表空间里去。

4) 重新启动MySQL服务器。

重新配置InnoDB表空间还有一种办法, 即先转储、再用新配置重新加载它。具体步骤如下:

1) 使用`mysqldump`程序转储所有的InnoDB数据表。

2) 关停MySQL服务器, 删除现有的InnoDB表空间、InnoDB日志文件及其各InnoDB数据表的`.frm`文件。

3) 按新配置方案去重建InnoDB表空间。

4) 把在第1步得到的转储文件重新加载到MySQL服务器里以重新创建各个InnoDB数据表。

### 11.5.6 优化MySQL服务器

MySQL服务器的某些参数（变量）会对其运行情况产生影响。如果这些参数的默认值不适合你的系统，完全可以根据具体情况去修改它们。比如说，如果内存比较充裕，就应该加大MySQL服务器用于磁盘读写和索引操作的缓冲区尺寸。内存里容纳的信息越多，磁盘读写操作就会越少。反之，如果内存比较紧张，就应该减小缓冲区的尺寸；这往往会降低MySQL服务器的运行速度，但有助于改善系统的整体性能，因为其他进程肯定会受益于MySQL服务器退让出来的系统资源。

在接下来的几个小节里，将在以下几个方面展开讨论和介绍：1）MySQL服务器变量的查看和设置办法；2）一些普适性的MySQL服务器变量；3）一些专用于InnoDB数据表处理程序的MySQL服务器变量。MySQL服务器变量的完整清单可以在本书附录D对SHOW VARIABLES语句的介绍内容里找到。在MySQL Reference Manual（MySQL参考手册）与优化工作有关的章节里也有很多有价值的信息。

#### 1. 设置和查看MySQL服务器变量

MySQL服务器变量都可以在MySQL服务器启动时设定；在MySQL 4.0.3及以后的版本里，很多变量还允许在MySQL服务器保持运行时动态地进行修改。

用来在某个MySQL服务器启动时设置其服务器变量的语法随软件版本的不同而有所差异。在MySQL 4.0.2及以后的版本里，可以把一个变量名视为一个选项名并直接设置它的值。比如说，数据表缓存区的尺寸由服务器变量table\_cache控制，如果想把这个变量设置为128，可以在命令行里增加一个如下所示的“选项”：

```
--table_cache=128
```

也可以按如下所示的语法在某个选项文件中对这个变量进行设置：

```
[mysqld]
table_cache=128
```

这种“把变量视为选项”的语法还允许把变量名中的下划线（“\_”）写成连字符（“-”），让服务器变量看上去与真正的选项没有什么不同。比如下面这个命令行“选项”：

```
--table-cache=128
```

在选项文件里也可以把变量名中的下划线写成连字符：

```
[mysqld]
table-cache=128
```

还有一种办法是使用--set-variable或-O选项，可以像下面这样在命令行上使用它们：

```
--set-variable=table_cache=128
-O table_cache=128
```

也可以像下面这样在选项文件里使用它们（注意，在选项文件里只能使用选项的长格式）：

```
[mysqld]
set-variable=table_cache=128
```



如果需要设置多个变量，就要把每一个变量分别视为一个选项。

MySQL 4.0.2之前的版本不支持把变量名视为选项名的做法，只能通过--set-variable或-O选项来进行设置。MySQL 4.0.2及以后的版本仍支持--set-variable和-O选项，但还这样做的人已经很少了。

无论使用哪一种语法去设置MySQL服务器变量，把它们安排到选项文件里的做法往往更简明易用，因为你不必在每次启动MySQL服务器时都要记住去设置哪些变量了。

在MySQL 4.0.3之前的版本里，服务器变量只能在启动时设置，一旦MySQL服务器开始运行，就不允许再去改变服务器变量的值了。MySQL 4.0.3版本在服务器变量的处理方面引入了两项改进：

- 很多变量现在都允许人们在MySQL服务器仍在运行时进行设置了。这使MySQL服务器的工作情况更容易控制，需要关停MySQL服务器的事情也少多了。（比如说，当试验不同的缓冲区尺寸对MySQL服务器的性能有什么影响时，就不必再像从前那样每改变一次设置值就得关停并重新启动一次MySQL服务器了。）以这种方式设置的变量值其效力只能维持到MySQL服务器退出运行之时，因此，如果为某个变量试出了一个效果比其当前值更优的新设置值，就应该把新设置值写到某个选项文件里去，让MySQL服务器以后总是用新设置值进行启动。
- MySQL服务器变量的作用范围分大小两个级别——全局级和会话级。全局级变量将全面影响整个MySQL服务器，会话级变量则只影响某给定客户连接上的工作情况。如果某个变量同时存在于两个级别，MySQL将在客户建立连接的时候用全局级变量的值去初始化相应的会话级变量，但在客户连接建立起来之后，对全局级变量所做出的修改将不会影响到相应的会话级变量的当前值。

给定一个全局级变量`var_name`，下面两种SET语句格式之一都能对它进行设置：

```
SET GLOBAL var_name = value;
SET @@GLOBAL.var_name = value;
```

类似地，针对会话级变量的SET语句也有两种格式，如下所示：

```
SET SESSION var_name = value;
SET @@SESSION.var_name = value;
```

不带级别限定符的SET语句修改的是会话级变量，如下所示：

```
SET var_name = value;
SET @@var_name = value;
```

可以用一条SET语句设置多个变量，但要用逗号把各变量赋值语句隔开，如下所示：

```
SET SESSION sql_warnings = 0, GLOBAL table_type = InnoDB;
```

凡是允许出现关键字SESSION的地方，都可以用它的同义词LOCAL来替换（这包括用@@LOCAL来替换@@SESSION）。

必须具备SUPER权限才能对全局级变量进行设置，新设置值的效力将持续到该变量被再次修改或MySQL服务器退出执行。对会话级变量进行设置不需要特殊的权限，新设置值的效力将



持续到该变量被再次修改或当前连接被断开。

MySQL服务器变量的当前值可以用SHOW VARIABLES语句来查看。可以让这条语句列出所有的变量，也可以让它只显示变量名与给定SQL模式相匹配的变量：

```
SHOW VARIABLES;  
SHOW VARIABLES LIKE 'pat';
```

MySQL 4.0.3及更高版本的SHOW VARIABLES语句增加了几种格式，现在可以让它只列出全局级或者会话级变量的当前值：

```
SHOW GLOBAL VARIABLES;  
SHOW GLOBAL VARIABLES LIKE 'pat';  
SHOW SESSION VARIABLES;  
SHOW SESSION VARIABLES LIKE 'pat';
```

如果不带GLOBAL或SESSION关键字，这条语句将返回会话级变量的值，如果会话级变量不存在，则返回全局级变量的值。

在命令行上，mysqladmin variables将把MySQL服务器的全局级变量的当前值列出来。

在附录D里，在介绍SHOW VARIABLES语句时对MySQL服务器变量也做了详细的介绍，比如哪些变量允许动态修改、允许在哪个级别进行修改等等。

## 2. 通用的MySQL服务器变量

一些变量对一般的性能调整是最有用的，如下所述：

- **back\_log** 在MySQL服务器正在处理一个连接请求的时候，如果又有其他客户（程序）发来连接请求，那些后到的连接请求将排入一个队列等待MySQL服务器进行处理。这个变量给出的是这个队列所能容纳的待处理连接请求的最大个数。如果你的站点非常繁忙，就需要加大这个变量的值。
- **delayed\_queue\_size** 在被实际插入到各有关数据表里去之前，来自INSERT DELAYED语句的数据行将在一个队列里等待MySQL来处理它们；**delayed\_query\_size**就是这个队列所能容纳的数据行的最大个数。当这个队列满时，后续的INSERT DELAYED语句将被阻塞直到这个队列里有容纳它们的空间为止。如果有很多客户发出INSERT DELAYED语句且你发现这些语句有阻塞的迹象，加大这个变量的值将使更多的INSERT DELAYED语句更快地得到处理。（对INSERT DELAYED语句的详细介绍见第4.5节。）
- **flush\_time** 自动存盘时间间隔。如果你的系统经常死机或重启，把这个变量设置为一个适当的非零值将使MySQL服务器每隔flush\_time秒就去“冲刷”一次数据表缓存区（把其中的信息写入磁盘）。这将导致系统性能下降，但减少了数据表被破坏或者丢失数据的概率。在命令行上用--flush选项启动MySQL服务器将使它负责管理的数据表在每次修改操作后被自动存盘。
- **key\_buffer\_size** 用来容纳索引块的缓冲区的长度。加大这个变量的值能加快索引创建操作和索引修改操作的速度，进而加快基于索引的检索和排序操作的速度。索引缓冲区越大，MySQL在内存里找到键值的可能性就越大，在对索引进行处理时需要去读写磁盘的次数就越少。

在MySQL 3.23之前的版本里，这个变量叫做key\_buffer；在3.23版本及以后的版本里，这两个名字MySQL服务器都能识别。

- max\_allowed\_packet MySQL服务器在与客户（程序）之间进行通信时使用的缓冲区的最大长度。在MySQL 4之前的版本里，这个变量的最大可取值为16MB；在MySQL 4及以后的版本里，它的最大可取值是1GB。

如果你的客户经常批量传输一些尺寸较大的BLOB或TEXT值，这个服务器变量的值就可能需要加大。客户端的缓冲区长度（由一个同名的客户端变量控制，目前的默认长度是16MB）也应该相应地加大。比如说，如果想让mysql客户程序以64MB作为其数据包上限，就要用下面这条命令去启动它：

```
% mysql --set-variable=max_allowed_packet=64M
```

- max\_connections MySQL服务器允许同时处于打开状态的客户连接的最大个数。如果MySQL服务器很繁忙，可能需要加大这个值。比如说，假设MySQL服务器被一个繁忙的Web服务器用来处理大量的由DBI或PHP脚本生成的数据库查询命令，而你把这个变量设置得很小，网站的访问者们就会发现他们的请求经常被拒绝。
- table\_cache 数据表缓存区的尺寸。加大这个值将使mysqld能够同时打开更多的数据表，从而减少文件打开/关闭操作的次数。

如果加大了max\_connections或table\_cache变量的值，MySQL服务器就会占用更多的文件描述符，如果操作系统对每个进程所能占用的文件描述符的个数有限制，这样做就会引起一些问题。如果真的遇到这种问题，就需要加大操作系统对每个进程所能占用的文件描述符的个数限制或者另想办法绕过这一限制。有好几种办法能加大MySQL服务器所能占用的文件描述符的个数。首先，如果是用mysqld\_safe脚本去启动MySQL服务器的，可以在执行这个脚本时利用它的--open-files-limit选项来加大MySQL服务器所能占用的文件描述符的个数。其次，可以重新配置系统（有些系统比较容易配置，只要编辑它的系统定义文件再重新开机就能达到目的；另外一些系统则要复杂一些，可能需要编辑一个内核说明文件并重新编译内核才能奏效）；具体做法请参考系统的有关文档。

还有一种办法能绕开操作系统对每个进程所能占用的文件描述符的个数限制：把原本只有一个的MySQL数据目录分割成多个并运行多个MySQL服务器。实际上，这等于是把每个进程可用的文件描述符个数与运行MySQL服务器的个数相乘。但这个办法做起来比较复杂，并且很可能会带来一些其他问题——比如说，因为某个给定的MySQL服务器只能访问它自己的数据目录，所以将不能通过它去访问其他数据目录里的数据库；再比如说，如果想让用户能够访问多个MySQL服务器，就必须把他们的权限依次复制到各有关MySQL服务器的权限表里去。将在第11.6节对此做进一步讨论。

有些MySQL服务器变量是用来把系统资源分配给每一个MySQL客户程序的——如果你的系统经常有大量的MySQL客户程序在同时运行，加大它们的值就会导致MySQL服务器的资源占用总量急剧增加。比如说，为改善MySQL访问性能，MySQL管理员可能会加大MySQL服务器变量read\_buffer\_size和sort\_buffer\_size（这两个变量在MySQL 4.0.3之前的版本里分别叫做record\_buffer和sort\_buffer；前者负责设置关联查询操作所使用的缓存区的尺寸，后者负责设置

排序操作所使用的缓存区的尺寸)的值,但因为MySQL会给每个客户连接都分配这两个缓存区,所以如果把这两个变量的值设置得过大的话,性能反而会因为急剧增加的系统资源消耗量而受到损害。因此,在改变这种“每个连接都会有”的缓存区的尺寸时一定要谨慎从事。应该逐步加大有关变量的值并在修改生效后立刻测试其效果,千万不要一下子把它们设置得太大。只有这样,才能在不严重损害系统性能的前提下给有关变量设置一个最适当的值。还要注意的是一定要按实际情况去进行测试。这类缓存区都是“按需分配”而不是客户连接刚一建立就分配的(比如说,如果某个客户没有进行关联查询,MySQL就不会为它分配供关联查询用的缓存区),所以用来进行测试的客户(程序)和查询命令必须与实际情况保持一致才能让你掌握对有关变量的修改在MySQL服务器上的真实效果。

### 3. InnoDB处理程序变量

除通用的MySQL服务器变量外,如果激活了InnoDB支持,MySQL服务器就会有一些与InnoDB有关的变量。人们经常通过以下几个MySQL服务器变量来控制InnoDB处理程序的工作情况:

- `innodb_buffer_pool_size` 如果你有足够的内存,把这个变量设置得大一些能减少MySQL到硬盘上去读写数据表数据和索引的活动。
- `innodb_log_buffer_size` 加大这个缓冲区的尺寸将使更多的语句在你提交某个事务之前被缓存在内存里,从而减少了事务执行期间的磁盘读写次数。
- `innodb_log_file_size`和`innodb_log_files_in_group` 当日志被写满时,InnoDB处理程序将把其缓冲区池里的信息写到磁盘上去——InnoDB日志文件越大,就需要经过更长的时间才能被填满,InnoDB处理程序把其缓冲区池里的信息写到磁盘上去的次数也就越少。(不过,日志文件越大,数据库崩溃后的恢复工作就需要花费更多的时间才能完成。)修改`innodb_log_file_size`变量将改变InnoDB日志文件的长度,修改`innodb_log_files_in_group`变量将改变InnoDB日志文件的个数。InnoDB日志文件的总长度(即`innodb_log_files_in_group`变量值和`innodb_log_file_size`变量值的乘积)是一个十分重要的指标。但要提醒大家注意的是,InnoDB日志文件的总长度不得超过4GB。

## 11.6 运行多个MySQL服务器

大多数人只在一台给定的机器上运行一个MySQL服务器,但有些场合可能需要运行多个MySQL服务器:

- 在现有的MySQL服务器仍保持运转的情况下对MySQL服务器的某个新版本进行测试。也就是说,想同时运行多个不同版本的MySQL服务器程序。
- 很多操作系统都对单个进程所能使用的文件描述符的个数有限制。如果无法在操作系统级加大这个上限值,同时运行同一MySQL服务器的多个实例是一个绕开这一限制的好办法。在操作系统级加大这个上限值往往需要重新编译操作系统的内核,可如果你不是操作系统级的系统管理员的话,就不具备进行这项工作的权限。
- 因特网服务提供商(Internet Service Provider, ISP)大都会为自己的每一位重要顾客分别提供一套专用的MySQL安装,而这就意味着要同时运行多个MySQL服务器。如果顾客们

选用的都是同一版本的MySQL软件，ISP就需要同时运行同一MySQL服务器程序的多个实例；如果顾客们选用的是不同版本的MySQL软件，ISP就需要同时运行多个不同版本的MySQL服务器程序。

以上是一些较为常见的需要运行多个MySQL服务器的理由，但其他一些场合可能也需要这么做。比如说，如果你想写一本关于MySQL的书，就需要对各种版本的MySQL服务器做全面的测试以了解它们之间的行为差异。笔者的情况就属于这一类——我在自己的机器上安装了30多种不同版本的MySQL服务器。不过，我总在使用的只是其中的一两种，其他的只是在需要对之进行测试时才偶尔运行一次，所以我得设法让自己随时都能方便地启动和关停它们才行。

### 11.6.1 运行多个MySQL服务器需要注意的问题

运行多个MySQL服务器要比只运行一个复杂得多，因为必须防止它们互相干扰。在安装MySQL软件时需要注意的问题是：要想使用多个不同的版本，就必须把它们分别安装到不同的位置。对于预编译好的二进制发行版本，只需在不同的目录里对它们进行解压缩就能做到这一点；对于需要由你去自行编译的源代码发行版本，可以利用configure脚本的--prefix选项为不同的发行版本分别设定一个不同的安装路径。

在启动MySQL服务器时需要注意的问题是：每个MySQL服务器进程多少总会有几个与众不同的参数值。比如说，不管同时运行的是同一MySQL服务器的多个版本还是多个不同版本的MySQL服务器，不同的MySQL服务器进程必须监听不同的TCP/IP端口，否则它们就会相互冲突。类似问题还会出现在激活了日志功能的时候：每一个MySQL服务器都应该把它自己的日志信息写到它自己的日志文件里去，如果几个MySQL服务器往同一组日志文件里写东西，就肯定会出问题。

MySQL服务器的启动选项可以在启动时通过选项文件或命令行参数实时地给出；如果将运行的多个MySQL服务器程序都是你本人从它们的源代码编译出来的，也可以由你在它们的编译阶段给它们分别设定一组不同的参数。在编译阶段设定的参数将成为各MySQL服务器的内建默认值，在启动它们时就用不着再明确地给出它们了。

在运行多个MySQL服务器的时候，一定要把它们各自的参数都详细地记录下来以免弄混或者忘记。用选项文件来设定各有关参数能帮你记住它们。（即使你已经把各个MySQL服务器与众不同的参数值都编译到它们各自的二进制代码里了，再在它们各自的选项文件里把有关参数列举一遍也很有实用价值——可以把选项文件当做一种帮助文档来用。）

下面列举的几个选项都是比较容易引起冲突的，应该为不同的MySQL服务器给它们设置不同的值。注意，其中有几个选项会影响到其他选项，所以可能用不着为每一个MySQL服务器都明确地把下面列举的选项都设置一遍。比如说，每个MySQL服务器在运行时都会生成一个独一无二的进程ID（process ID，PID）文件，但因为PID文件的默认存放地点是相应的MySQL数据目录，所以如果你已经为不同的MySQL服务器设定了不同的数据目录，就用不着再明确地为它们的PID文件设定一个默认存放地点了。

- 如果所运行的MySQL服务器版本各不相同，通常是把不同的版本安装在不同的基本安装



路径下,而且最好是让不同的MySQL服务器使用不同的数据目录<sup>①</sup>。要想明确地设定这些值,请使用以下选项:

选 项	用 途
--basedir=dir_name	MySQL软件的安装路径名
--datadir=dir_name	MySQL数据目录的路径名

一般说来,某给定MySQL服务器的数据目录应该是其安装路径下的一个子目录,但不一定总是这样。比如说,很多ISP都为自己的顾客准备了一个公用的MySQL安装路径以节约空间(也就是说,顾客们使用的MySQL服务器程序和客户程序都是相同的),但会为每位顾客分别运行一个不同的MySQL服务器实例并让不同的MySQL服务器实例去管理不同顾客的数据目录。此时,所有MySQL服务器的基本安装路径都是一样的,但它们所负责管理的数据目录却各不相同——比如说,分别放在各位顾客的登录目录(home directory)下。

- 为防止多个MySQL服务器相互干扰,下列选项对于不同的MySQL服务器必须有不同的设置值:

选 项	用 途
--port=port_num	供TCP/IP连接使用的端口号
--socket=file_name	UNIX域套接字文件的路径名
--pid-file=file_name	给定MySQL服务器的PID文件的路径名

- 如果激活了日志功能,就必须让不同的MySQL服务器使用不同的日志文件名,否则就会出现多个MySQL服务器都把自己的日志信息写到同一个日志文件里去的混乱局面。往好处说,你的日志文件将没几个人能看明白;往坏处说,建立镜像机制之类的工作将根本无法正确进行。在用下列选项设定日志文件名的时候,如果给出的是相对文件名,日志文件将被创建在相应的MySQL服务器的数据目录里。如果已经为不同的MySQL服务器指定了不同的数据目录,就用不着使用绝对路径名来设定这些日志文件的名字了(日志文件的命名规则见第11.4节)。

日 志 选 项	该选项激活的日志
--log[=file_name]	常规日志文件
--log-bin[=file_name]	二进制变更日志文件
--log-bin-index=file_name	二进制变更日志的索引文件
--log-update[=file_name]	变更日志文件
--log-slow-queries[=file_name]	慢查询日志文件
--log-isam[=file_name]	ISAM / MyIASM日志文件

- 如果激活了BDB或InnoDB数据表处理程序,就必须让不同的MySQL服务器把它的BDB或InnoDB日志写到不同的目录里去。在默认的情况下,某给定MySQL服务器将把其BDB或InnoDB日志写到数据目录里去,但这个位置可以用下列选项改变:

<sup>①</sup> 把不同版本的MySQL服务器安装在同一安装路径下并不是绝对不可以,但不推荐这样做。



日志选项	用途
<code>--bdb_logdir=dir_name</code>	用来存放BDB日志文件的目录
<code>--innodb_log_arch_dir=dir_name</code>	用来存放InnoDB日志档案的目录
<code>--innodb_log_group_home_dir=dir_name</code>	用来存放InnoDB日志文件的目录

只要设定了两个InnoDB选项中的某一个,就必须对另一个也进行设定,而且它们的值必须相同。

- 在UNIX系统上,如果是用mysql\_safe脚本去启动MySQL服务器的,这个脚本将创建一个错误日志(该日志的默认存放位置是相应的MySQL数据目录)。可以用`--err-log=file_name`选项给这个日志明确地设定一个文件名。但要提醒大家注意两件事:1)必须把这个选项传递给mysqld\_safe脚本而不是mysqld程序;2)如果给出的是一个相对路径名,MySQL服务器将从调用mysqld\_safe脚本时所在的目录开始对这个相对路径名做出解释而不是像对待其他日志文件那样从自己的数据目录开始做出解释。因此,如果打算使用`--err-log`选项,就应该给出一个绝对路径名以确保错误日志总是被创建在同一个地点。
- 在UNIX系统上,可能还需要用`--user`选项为不同的MySQL服务器指定一个不同的登录账户来运行之。这与你为不同用户分别提供一个MySQL服务器实例并让他们各自“拥有一个MySQL数据目录”的做法在原理上是相通的。
- 在Windows系统上,被安装为Windows服务的多个MySQL服务器必须有不同的服务名。

### 11.6.2 配置和编译不同的MySQL服务器

如果打算建立不同版本的MySQL服务器,应该把它们安装到不同的地点。要想让安装在同一台机器里的多套MySQL软件不相互干扰,最简单的方法是在运行其编译配置脚本configure时用`--prefix`选项为它们分别设定一个基本安装路径;要是再把版本号加到基本安装路径的名字里,就更容易区分哪个安装路径对应着哪一个MySQL版本了。下面将介绍一种具体的办法——我就是用这个办法在我自己的机器上安装了多套MySQL软件而又不让它们相互干扰的。

把各个版本的MySQL软件统一安装在目录`/var/mysql`下,把某给定版本的MySQL软件安装在`/var/mysql`目录下一个以其发行版本号为名的下级子目录里。比如说,在安装MySQL 4.0.5版本时使用的基本安装路径是`/var/mysql/40005`,所以在运行configure脚本时就要用一个`--prefix=/var/mysql/40005`选项来设定之。还用了其他几个选项为不同版本的MySQL软件设定诸如TCP/IP端口号、套接字路径名之类的专用参数——把某给定MySQL服务器的TCP/IP端口号设置为它的版本号,把它的套接字文件直接放入其基本安装目录,再把它的数据目录指定为其基本安装路径下的`data`目录。

为了快速设置这些配置选项,编写一个名为`config-ver`的shell脚本,其框架如下(请注意,configure脚本是用`--localstatedir`而不是`--datadir`选项来为MySQL服务器设定数据目录的):

```
VERSION="40005"
PREFIX="/var/mysql/$VERSION"
# InnoDB is included by default as of MySQL 4:
# - prior to 4.x, include InnoDB with --with-innodb
# - from 4.x on, exclude InnoDB with --without-innodb
```

```

HANDLERS="--with-berkeley-db"
OTHER="--enable-local-infile --with-embedded-server"
rm -f config.cache
./configure \
    --prefix=$PREFIX \
    --localstatedir=$PREFIX/data \
    --with-unix-socket-path=$PREFIX/mysql.sock \
    --with-tcp-port=$VERSION \
    $HANDLERS $OTHER

```

这样，只要在config-ver脚本的第一行给出正确的版本号，再根据是否需要使用InnoDB或BDB数据表处理程序、是否需要使用LOAD DATA语句的LOCAL能力等情况修改相应的配置选项值，就可以用下面几条命令迅速完成某给定MySQL发行版本的配置、编译和安装工作了：

```

% sh config-ver
% make
% make install

```

接下来，把路径切换到这个MySQL发行版本的基本安装目录并对它的数据目录和权限表进行初始化：

```

% cd /var/mysql/40005
% ./bin/mysql_install_db

```

接着，按照第11.2节以及第12章中的有关步骤对新安装的MySQL软件进行必要的处理。

完成上述工作后，剩下的事情就是确定这个MySQL服务器将使用哪些选项来启动以及如何编写其选项文件了。将在第11.6.4节对此做详细介绍。

### 11.6.3 设定MySQL服务器启动选项的策略

把多个MySQL服务器都安装好以后，怎样才能正确地让它们使用各自的启动选项去启动和运行呢？有以下几种选择：

- 如果将启动运行的MySQL服务器都是你自行编译的，可以在编译阶段把它们各自的默认启动选项编译到各个MySQL服务器程序里去。这种做法的好处是在启动各MySQL服务器时不必再给出任何选项，坏处是你可能会忘记各MySQL服务器都是用哪些选项启动的。
- 如果想在启动时再给出各个MySQL服务器的启动选项，有命令行选项或选项文件两种办法可供选择。如果需要设定的选项很多，在命令行上列出它们的做法就比较麻烦。把启动选项放到选项文件里的做法更简明易行，但需要保证各MySQL服务器所读取的选项都是正确无误的。下面是完成这项工作的几种策略：
  - 用--defaults-file选项指定一个文件，让MySQL服务器到这个文件里去读取它所有的选项。这需要为每一个MySQL服务器分别指定一个不同的文件。这种做法的好处是可以把某给定MySQL服务器在启动时需要用到的所有选项全都放在一个地方，而它在启动时只需读取一个选项文件就行了。（注意，一旦使用了--defaults-file选项，MySQL服务器将不再读取/etc/my.cnf等“正常的”选项文件。）

- 先把所有MySQL服务器都要用到的公共选项放到一个全局级选项文件（比如/etc/my.cnf）里去，再在命令行上用--defaults-extra-file选项为某给定MySQL服务器指定一个附加的选项文件。比如说，如果把适用于所有MySQL服务器的选项统一安排在/etc/my.cnf文件中的[mysqld]选项组里，就用不着在各个MySQL服务器专用的选项文件里重复列举它们了。

这里要提醒大家注意的是，放在公共选项组里的选项必须是打算运行的各个MySQL服务器都能支持的。比如说，如果打算运行的某个MySQL服务器的版本早于3.23.29，就不能在公共选项组里用--local-infile选项来激活LOAD DATA语句的LOCAL能力。这是因为，--local-infile选项在早于3.23.29版本的MySQL软件里不存在，把它放到公共选项组里会导致老版本的MySQL服务器启动失败。

- MySQL服务器会到你在其编译阶段设定的数据目录位置去寻找一个名为my.cnf的选项文件。如果已经在各个MySQL服务器的编译阶段为它们分别设定了一个数据目录路径名，就可以把它们各自的专用选项分别列举在这些my.cnf文件里。也就是说，可以把全体MySQL服务器都要用到的所有公共选项统一集中到/etc/my.cnf文件里，再把各个MySQL服务器专用的选项分别写到相应的DATADIR/my.cnf（DATADIR代表各个MySQL服务器的数据目录）文件里去。（注意：如果在启动某给定MySQL服务器时给它另外指定了一个数据目录位置，这个策略就不管用了。这个办法也不适用于将同时运行某给定MySQL服务器程序的多个实例的情况。）
- 用mysqld\_multi脚本去启动多个MySQL服务器。这个脚本允许把多个MySQL服务器的选项安排在同一个选项文件里，每一个MySQL服务器分别对应于该文件里的一个选项组。

- 在Windows系统上，还可以采用把它们运行行为多项Windows服务的办法来启动多个MySQL服务器。此时，需要按照一组专门为此制定的命名规则去给选项文件起名。

下面，我们一起来看看如何使用mysqld\_multi脚本以及如何在Windows系统上运行多个MySQL服务器。

#### 11.6.4 用mysqld\_multi脚本来启动多个MySQL服务器

在UNIX系统上，如果只运行一个MySQL服务器，用mysqld\_safe和mysqld\_server脚本来启动它将最合适；可如果想启动多个MySQL服务器，那最好还是使用mysqld\_multi脚本来进行。

mysqld\_multi脚本的具体用法是：先为打算启动的每一个MySQL服务器分配一个编号，再把它启动选项写入某个选项文件的[mysqldn]（n是你分配给它的编号）选项组；mysqld\_multi本身要用到的选项可以列在这个选项文件中的[mysqld\_multi]选项组里。比如说，如果将安装MySQL 3.23.51、4.0.5和4.1.0等三个版本的MySQL服务器，则为选项组起名为[mysqld32351]、[mysqld40005]和[mysqld40100]，然后在选项文件/etc/my.cnf文件里做出以下设置：

```
[mysqld32351]
basedir=/var/mysql/32351
datadir=/var/mysql/32351/data
```

```

mysqld=/var/mysql/32351/bin/mysqld_safe
socket=/var/mysql/32351/mysql.sock
port=32351
local-infile=1
user=mysqladm
log=log
log-update=update-log
innodb_data_file_path = ibdata1:10M

[mysqld40005]
basedir=/var/mysql/40005
datadir=/var/mysql/40005/data
mysqld=/var/mysql/40005/bin/mysqld_safe
socket=/var/mysql/40005/mysql.sock
port=40005
local-infile=1
user=mysqladm
log=log
log-bin=binlog
innodb_data_file_path = ibdata1:10M:autoextend

[mysqld40100]
basedir=/var/mysql/40100
datadir=/var/mysql/40100/data
mysqld=/var/mysql/40100/bin/mysqld_safe
socket=/var/mysql/40100/mysql.sock
port=40100
local-infile=1
user=mysqladm
log=log
log-bin=binlog
skip-innodb
skip-bdb
language=french
default-character-set=utf8

```

这里给出的配置参数对应于第11.6.2节里介绍的目录布局。同时还对各MySQL服务器的套接字文件、TCP/IP端口号、日志、数据表处理程序等进行了设置。

要想启动某给定MySQL服务器，在mysqld\_multi脚本的命令行上给出命令字start和这个MySQL服务器的编号即可，如下所示：

```
% mysqld_multi --no-log start 32351
```

--no-log选项的作用是让MySQL把状态信息发送到控制台终端而不是日志文件，这样，就能从控制台终端的屏幕上直接观察到启动过程的进展情况了。可以用一个mysqld\_multi命令行启动多个MySQL服务器，但要用逗号把各服务器的编号彼此隔开。服务器编号还允许以一个区间的形式给出，区间的起止编号要用连字符来分隔。但要特别注意的是，编号之间绝对不允许出现

空格。如下所示：

```
% mysqld_multi --no-log start 32351,40005-40100
```

如果想关停某个MySQL服务器或者想查看一下它是否在运行，在mysqld\_multi脚本的命令行上给出命令字stop或report、再写出相应的服务器编号即可。对于这些命令，mysqld\_multi脚本将调用mysqladmin程序去与MySQL服务器通信，所以还需要给出一个管理员账户的用户名和口令才行。如下所示：

```
% mysqld_multi --nolog --user=root --password=rootpass stop 32351
% mysqld_multi --nolog --user=root --password=rootpass report 32351,40100
```

所给出的用户名和口令必须能让你在各有关MySQL服务器上执行有关操作。mysqld\_multi脚本会去自动寻找mysqladmin程序的位置，但也可以把这个路径明确地写在某个选项文件的[mysqld\_multi]选项组里。还可以把用来执行stop和report命令的管理员用户名和口令放到[mysqld\_multi]选项组里去，如下例所示：

```
[mysqld_multi]
mysqladmin=/usr/local/mysql/bin/mysqladmin
user=leeloo
password=multipass
```

注意，如果把管理员用户名和口令放到了某个选项文件里，千万记住把该文件的访问权限设置成不允许其他账户进行读操作！

#### 11.6.5 在Windows系统上运行多个MySQL服务器

在Windows系统上运行多个MySQL服务器有两个方法：一是以手动方式启动它们；二是把它们运行为多个Windows服务。

如果想以手动方式启动多个MySQL服务器，就需要为它们分别创建一个选项文件并把相应的各有关参数写到这个文件里去。比如说，如果想运行同一MySQL服务器程序的两个实例但让它们各有各的数据目录，就需要写出类似于下面这样的两个选项文件来：

C:\my.cnf1文件：

```
[mysqld]
basedir=C:/mysql
datadir=C:/mysql/data
port=3306
```

C:\my.cnf2文件：

```
[mysqld]
basedir=C:/mysql
datadir=C:/mysql/data2
port=3307
```

然后，从命令行启动MySQL服务器的时候，用--defaults\_file选项表明想使用哪一个选项文件：



```
C:\> mysqld --defaults-file=C:\my.cnf1
C:\> mysqld --defaults-file=C:\my.cnf2
```

这两个MySQL服务器启动之后，客户程序（包括用来关停MySQL服务器的mysqladmin程序在内）需要通过指定端口才能建立与它们的连接。下面的第一条命令使用的是默认端口（3306），而第二条命令明确地指定了端口3307：

```
C:\> mysqladmin -p -u root shutdown
C:\> mysqladmin -P 3307 -p -u root shutdown
```

在基于Windows NT的系统上，从MySQL 4.0.2版本开始，可以把MySQL服务器安装为一项服务并在安装时给它起一个服务名<sup>①</sup>，如下所示：

```
C:\> mysql-nt --install service_name
```

这意味着可以把多个MySQL服务器用不同的服务名把它们安装为一系列不同的服务，具体做法如下：

- 如果不带service\_name参数，被安装为一项服务的MySQL服务器将使用默认服务名MySql，并去读取选项文件中的[mysqld]选项组。
- 如果带service\_name参数，被安装为一项服务的MySQL服务器将使用给定的服务名，并去读取选项文件中的[service\_name]选项组。
- 以默认服务名MySql运行的MySQL服务器将支持一个名为MySql的命名管道。而明确地给出了一个服务名的MySQL服务器将只监听TCP/IP连接而不支持命名管道——除非还用--socket选项明确地给出了一个套接字文件路径。
- 各个MySQL服务器所管理的数据目录必须不同。

我们来看一个例子。假设想运行两个mysql-nt实例，想让它们分别以MySql和mysqlsvc2为服务名和命名管道名并继续沿用上例中的数据目录布局。在某个标准的选项文件（比如C:\my.cnf）里为这两个MySQL服务器设置如下所示的选项：

```
# group for default (MySql) service
[mysqld]
basedir=C:/mysql
datadir=C:/mysql/data
port=3306
enable-named-pipe

# group for mysqlsvc2 service
[mysqlsvc2]
basedir=C:/mysql
datadir=C:/mysql/data2
port=3307
enable-named-pipe
socket=mysqlsvc2
```

这样，就可以用下面这些命令去安装和启动相应的服务了：

<sup>①</sup> MySQL 3.23.54以后的3.23系列版本也允许设定一个服务名参数。

```

C:\> mysql-nt --install
C:\> net start MySql
C:\> mysql-nt --install mysqlsvc2
C:\> net start mysqlsvc2

```

这两个MySQL服务器启动之后,使用默认端口或默认命名管道名的客户程序将连接到默认的MySQL服务器;如果想连接到第二个MySQL服务器,就需要明确地给出其端口号或命名管道名,如下所示:

```

C:\> mysql --port=3307
C:\> mysql --host=. --socket=mysqlsvc2

```

如此启动的MySQL服务器可以用mysqladmin shutdown命令、net stop命令或Windows的“Services Manager”(服务管理器)来关停。如果想卸载被安装为一项Windows服务的MySQL服务器,先关停之(如果它仍在运行的话),然后用--remove命令字和你给它起的服务名来卸载它,如下所示:

```

C:\> mysql-nt --remove
C:\> mysql-nt --remove mysqlsvc2

```

在MySQL 4.0.3及以后的版本里,还可以在安装MySQL服务器时在命令行的最末尾使用一个--defaults-file选项,如下所示:

```

C:\> mysqld-nt --install service_name --defaults-file=file_name

```

这等于是提供了另一种设定MySQL服务器专用选项的手段。MySQL服务器将记住用--defaults-file选项给出的文件名并在启动过程中读取其内容,MySQL服务器将从这个文件的[mysqld]选项组去读取选项。如果打算使用这种服务安装语法,就必须给出服务名——即便想使用的是默认服务,也要明确地给出服务名MySQL。

## 11.7 设置镜像服务器

建立数据库“镜像”的办法之一是把某原始数据库拷贝到另一个MySQL服务器去。可是,如果这个原始数据库的内容以后又发生了变化而你又想让它的复制品及时跟上这些变化,就不得不在复制品上重复各有关操作。要想让复制品能够在原始数据库的内容发生变化时及时做出相应的修改,可使用MySQL的实时镜像机制。这一机制使我们能够随时拥有原始数据库的一份拷贝,并在原始数据库的内容发生变化时让它们自动而且及时地在复制品上反映出来。

### 11.7.1 镜像机制概念

MySQL数据库系统中的镜像机制遵循以下原则:

- 在镜像关系中,一个MySQL服务器将扮演“主人”角色(即所谓的“主MySQL服务器”,以下简称“主服务器”),另一个则扮演“奴隶”角色(即所谓的“从MySQL服务器”,以下简称“从服务器”);“奴隶”将严格按照“主人”的一举一动行事。必须给这两个MySQL服务器分别分配一个独一无二的镜像ID。

- 在镜像关系形成之初，主服务器与从服务器必须完全同步——即该镜像关系所涉及的各有关数据库在这两个MySQL服务器上必须有着完全一样的内容。在镜像关系形成之后，人们在主服务器上做出的修改动作将被传输并实现在从服务器上，但人们在主服务器上做出的修改动作并不直接作用在从服务器上的镜像数据库上。
- 负责在主、从服务器间传输各种修改动作的媒介是主服务器的二进制变更日志，这个日志记载着需要传输给从服务器的各种修改动作。因此，主服务器必须激活二进制日志功能。
- 从服务器必须具备足以让它连接主服务器并请求主服务器把其二进制变更日志传输给它的权限。人们将通过“镜像协调信息”来掌握从服务器上的进展情况，“镜像协调信息”由从服务器当前正在读取的二进制变更日志文件名和它在该文件里的当前读写位置构成。
- 每个主服务器可以有多个从服务器，但每个从服务器只能有一个主服务器。不过，MySQL允许把一个从服务器用做另一个从服务器的主服务器，也就是说，可以创建一个镜像服务器链。

相对而言，镜像机制在MySQL软件里还是一个新生事物，人们仍在对它进行着开发和完善，所以想及时掌握又新增了哪些镜像功能并不是件容易的事。镜像机制最早出现于MySQL 3.23.15版本。一般来说，建议大家尽可能使用最新的版本，但在做出最终决策的时候，不同版本的MySQL服务器在镜像能力方面的局限性也是一个必须考虑的因素。下面是不同版本的MySQL服务器在镜像机制方面的兼容规则：

- 3.23.x系列版本的从服务器不能与4.x系列版本的主服务器通信。
- 4.0.0版本的从服务器只能与4.0.0版本的主服务器通信。
- 4.0.1或更高版本的从服务器既能与3.23.x系列版本的主服务器通信，也能与4.x系列版本的主服务器通信。但后一种情况要求主服务器的版本号等于或大于从服务器的版本号。

一般说来，建议大家遵循以下原则：

- 要尽可能地让主服务器和从服务器都来自同一MySQL版本系列。比如说，应该选用3.23.x版本的从服务器去配合3.23.x版本的主服务器，不要选用3.23.x版本的从服务器去配合4.x版本的主服务器——反之亦然。
- 在选定3.23.x或4.x版本系列后，尽量使用该系列中的最新版本。相对而言，同一系列中的最新版本往往有着最丰富的功能、最少的局限性以及最少的程序漏洞。

镜像机制中的主、从服务器不仅要在版本上相互兼容，还需要在功能上相互兼容。比如说，如果主服务器里有使用了外键的InnoDB数据表和要求有RAID（磁盘阵列）支持的MyISAM数据表，从服务器就必须带有InnoDB处理程序和RAID支持。

### 11.7.2 建立主 - 从镜像关系

在两个MySQL服务器间建立主 - 从镜像关系的具体步骤如下：

- 1) 确定自己想给这两个MySQL服务器分配什么样的ID值。主、从服务器的ID值必须是彼此不同的64位正整数。在启动主、从服务器的时候，必须用server\_id启动选项给出其ID值。
- 2) 从服务器必须在主服务器上有一个具备足够权限的账户，从服务器将使用这个账户去连接主服务器并请求主服务器把二进制变更日志发送给它。可以用以下命令在主服务器上为从服

务器创建一个账户：

```
GRANT REPLICATION SLAVE ON *.*
TO 'slave_user'@'slave_host'
IDENTIFIED BY 'slave_pass';
```

上例中的REPLICATION SLAVE权限适用于MySQL 4.0.2及以后的版本，如果你使用的MySQL版本较早，请使用FILE权限。（MySQL 4.0.2对权限表做了较大的修改，很多权限——包括镜像机制中使用的FILE等权限在内——都发生了变化。）*slave\_user*和*slave\_pass*值将在稍后设置从服务器时用到。如果这个账户的用途仅限于镜像机制，就不需要再授予它其他权限了。不过，为了对镜像机制进行测试，通常要在从服务器上用mysql程序以“手动方式”去连接主服务器，所以往往还应该再给这个账户多授予几项权限。（比如说，如果这个账户仅有REPLICATION SLAVE权限的话，甚至无法在从服务器上用SHOW DATABASES语句去查看来自主服务器的数据库名。）

3) 把主服务器上的数据库拷贝到从服务器以完成主、从服务器之间最初的同步。一种办法是先在主服务器主机上制作一份备份、再把这个备份加载到从服务器上去；另一种办法是通过网络把各有关数据库从主服务器全部拷贝到从服务器。本书第13章对数据库的备份和拷贝技术做了详细的介绍。

在MySQL 4.0.0及以后的版本里，还可以用LOAD DATA FROM MASTER语句来建立从服务器，但这条语句要求满足以下几个条件：

- 这个镜像关系所涉及的数据表全都是MyISAM数据表。
- 为发出这条语句而在连接从服务器时使用的账户必须具备SUPER权限。
- 从服务器用来连接主服务器的账户必须具备RELOAD和SUPER权限。如果想在创建这个账户时把RELOAD和SUPER权限与REPLICATION SLAVE权限一起授予给它，就需要把刚才那个例子里的GRANT语句修改成如下所示：

```
GRANT REPLICATION SLAVE,RELOAD,SUPER ON *.*
TO 'slave_user'@'slave_host'
IDENTIFIED BY 'slave_pass';
```

请注意，这是一个主服务器上的账户，而用来发出LOAD DATA FROM MASTER语句的账户是一个从服务器上的账户，千万不要把它们弄混了。

- LOAD DATA FROM MASTER语句在执行时需要申请一个全局性的读操作锁，这个读操作锁将在LOAD DATA FROM MASTER语句执行期间阻塞主服务器上的一切写操作。

这些限制有可能在今后被放松或者彻底取消。

无论用哪一种办法把数据库从主服务器拷贝到从服务器，在开始制作备份到（如有必要的话）给主服务器重新配置好二进制日志功能的这段时间内，都必须确保在主服务器上没有任何修改（写）操作。

4) 关停主服务器（如果它正在运行的话）。

5) 对主服务器的配置进行修改——把它的镜像ID告诉它并激活其二进制日志功能。需要在主服务器启动时会去读取的某个选项文件里添加以下几行指令：



```
[mysqld]
server-id=master_server_id
log-bin=binlog_name
```

6) 重新启动主服务器；从现在起，它将把客户对有关数据库的修改记载到二进制变更日志里去。（如果此前已经激活了它的二进制日志功能，请在重新启动它之前把现有的二进制变更日志备份下来，等它重新启动后再发出一条RESET MASTER语句去清除现有的二进制变更日志。）

7) 关停从服务器（如果它正在运行的话）。

8) 对从服务器进行配置——以便让它知道自己的镜像ID、到哪里去找主服务器以及如何去连接主服务器。最简单的情况是主、从服务器分别运行在不同的主机上并都使用着默认的TCP/IP端口，只要在从服务器启动时会去读取的某个选项文件里添加以下几行指令就行了：

```
[mysqld]
server-id=slave_server_id
master-host=master_host
master-user=slave_user
master-password=slave_pass
```

*slave\_server\_id*是从服务器的镜像ID，它不得与主服务器的镜像ID相同。*master\_host*是主服务器的主机名。在UNIX系统上，如果主服务器和从服务器将运行在同一台主机上，就必须把这个主机名写成127.0.0.1而不是localhost才能确保从服务器使用一条TCP/IP连接。（以localhost为主机名将使从服务器使用一个套接字文件去连接主服务器，但镜像机制不支持经套接字文件建立的连接。）*slave\_user*和*slave\_pass*值是在主服务器上为从服务器创建的那个账户的用户名和口令，从服务器将使用这个账户去连接主服务器并请求主服务器把二进制变更日志传输给它。注意，应该把这几行指令添加到一个只有从服务器上的MySQL管理员登录账户才能访问的选项文件里，因为这个用户名和口令应该是保密的——比如说，不要把它们放在选项文件/etc/my.cnf里，因为它通常是对全体用户可读的。建议把这几行指令放到从服务器的数据目录中的my.cnf文件里去，并对这个数据目录进行安全处理；具体做法请参见本书第12章中的有关内容。

如果需要对从服务器与主服务器之间的连接做更具体的设置，还可以——如果主服务器所监听的网络端口不是默认端口的话——在从服务器的[mysqld]选项组里用master-port选项明确地写出主服务器所监听的网络端口。

如果主、从服务器之间的连接是间歇性的或者不太可靠，可能还需要修改“连接重试间隔”（默认值是60秒）和“连接重试次数”（默认值是86 400次）的默认值；可以用master-connect-retry和master-retry-count选项来设置这两个参数的值。

9) 重新启动从服务器。从服务器使用两个信息源来确定它自己在镜像工作中的进度位置：一个是其数据目录中的master.info文件，另一个是其启动选项所给定的配置信息。在第一次启动从服务器的时候，因为master.info文件尚不存在，所以从服务器将根据你在其选项文件中给出的各种master-xxx选项值去连接主服务器。连接成功之后，从服务器将创建一个新的master.info文件来保存各种连接参数和它自己的镜像工作状态。等以后再启动从服务器的时候，它将优先使用master.info文件而不是选项文件里的信息去连接主服务器。（这意味着如果你以后又在选项文件里修改了主服务器的主机信息，就必须删除master.info文件并重新启动从服务器；否则，所做



出的修改将无法生效。)

以上步骤适用于打算把某MySQL服务器上所有的数据库(包括容纳着各种权限表的mysql数据库在内)都镜像到另一个MySQL服务器上的情况。如果不想让主、从服务器有完全相同的账户信息(比如在打算建立一个私用的镜像从服务器并不想让在主服务器上有账户的人们连接到从服务器的时候),可以把mysql数据库排除在镜像机制外。要想把mysql数据库排除在外,需要在主服务器的选项文件中的[mysqld]选项组里加上下面这行指令,这样,当在主服务器上激活二进制日志功能后,mysql数据库上的操作就不会被记载到其二进制变更日志里了:

```
binlog-ignore-db=mysql
```

如果要把多个数据库排除在镜像机制外,就需要多次使用这个选项——每个数据库一次。

在建立起镜像关系并使之开始运转之后,还需要对主、从服务器进行监控和管理,这些工作可以用以下语句(对这些语句的详细介绍见本书的附录D,这里只是一个简单的介绍)来完成:

- **SLAVE STOP**和**SLAVE START**——用来挂起和恢复从服务器上的镜像活动。比如说,当制作备份时,就可以用这些语句让从服务器暂时停止镜像活动。
- **SHOW SLAVE STATUS**——在从服务器上查看其“镜像协调信息”。这些信息可以用来判断哪些二进制变更日志已经不再会被用到了。
- **PURGE MASTER**——在主服务器上对二进制变更日志进行失效处理。当在每一个从服务器上都使用**SHOW SLAVE STATUS**语句查看过它的“镜像协调信息”之后,可以在主服务器上用这条语句去删除那些已经不再会被用到的二进制变更日志文件。
- **CHANGE MASTER**——在从服务器上对它正在使用的几个镜像参数(比如,它正在读取主服务器的哪一个二进制变更日志、它正在写哪一个中继日志文件等等)进行修改。

在MySQL 4.0.2及以后的版本里,镜像机制中的从服务器是由两个内部线程实现的:一个叫“I/O线程”,负责与主服务器进行通信、请求主服务器发送二进制变更日志文件、把接收到的数据修改命令写入某个中继日志文件等等;另一个叫“SQL线程”,负责从中继日志中读出数据修改命令并执行之。(中继日志相当于“I/O线程”与“SQL线程”之间的通信纽带。)如果从服务器是用这种双线程模型实现出来的,就可以通过在**SLAVE STOP**或**SLAVE START**语句的末尾加上**IO\_THREAD**或**SQL\_THREAD**关键字的办法分别挂起或者恢复这两个线程中的任何一个。比如说,**SLAVE STOP SQL\_THREAD**命令将使从服务器停止执行中继日志里的数据修改命令,但从服务器仍能继续接收来自主服务器的数据修改命令并把它们记录到中继日志里去。

类似于二进制变更日志的情况,从服务器将把中继日志生成为一系列按数字编号的文件,而且也会有一个类似于二进制变更日志索引文件的中继日志索引文件。中继日志和中继日志索引文件的默认文件名分别是**HOSTNAME-relay-bin.nnn**和**HOSTNAME-relay-bin.index**,但可以用从服务器的启动选项**--relay-log**和**--relay-log-index**去改变这两个默认值。另一个与镜像机制有关的状态文件是中继信息文件,它的默认文件名是**relay-log.info**,可以用**--relay-log-info-file**选项改变这个文件名。

## 11.8 升级MySQL软件

MySQL软件第一次公开发行的版本是3.11.1。它目前有两个版本系列:一个是运行稳定的

4.0.x系列；一个是仍在不断开发中的4.1.x系列。这两个版本系列的升级版都发布得相当频繁，而这种持续不断的开发步伐让MySQL数据库系统的管理员们不得不面对这样一个问题：是否要在新版本发布时对现有的MySQL安装程序进行升级？为帮助大家做出正确的决定，将在下面的讨论内容里提出一些有关这方面的指导意见。

在有新版本出现的时候，应该做的第一件事是去了解它和以前的版本到底有哪些不同。如果为了让自己能在第一时间知道MySQL软件又推出了新版本而订阅了announce@lists.mysql.com邮件列表，你将在每一份新版本发布通告里找到对新增功能的介绍和说明。（或者，阅读MySQL发行版本自带的MySQL Reference Manual（MySQL参考手册）中的“Change Notes”（新增功能）附录也可以让你了解它都有哪些新增功能。）在知道新旧版本的差异之后，问自己下面几个问题：

- 新版本是否对你在使用当前版本的过程中遇到的问题进行了修正？
- 新版本是否有你想要或者需要的新增性能？
- 新版本是否能够改善你现有的某些操作的性能？

如果对这些问题的回答全都是“NO”，就没有必要去进行升级。如果你的回答里有“Yes”，就应该进行升级。不过，最好先等几天并到MySQL邮件列表上看看别人对这个新版本的评价。这个新版本真的有用吗？有没有程序漏洞或其他问题？

在决定是否要进行升级的时候，还应该考虑以下几个因素：

- 稳定系列里的新版本通常是用来修复程序漏洞而不是用来增加新功能的，所以在稳定系列里进行升级通常要比在开发系列里进行升级的风险小。（当然，如果你正使用着一个开发系列的MySQL服务器，可能根本就不在乎冒风险！）
- 在对MySQL进行了升级之后，可能还需要对一些用MySQL C客户程序开发库链接和编译出来的程序进行升级。比如说，在对MySQL进行了升级之后，可能需要用新版本的MySQL C客户程序开发库重新对Perl语言中的DBD::mysql模块或者对PHP语言解释器进行一次编译和链接。（如果你现有的与MySQL有关的DBI和PHP脚本在你升级MySQL之后都出现了异常行为，就往往预示着必须对DBD::mysql模块或PHP进行升级。）重新编译这些程序的工作量并没有多大，可如果你不想多费力气的话，就最好不要去升级MySQL。如果你使用的是静态链接程序而不是动态链接程序，这类问题的发生几率会小很多，但系统内存需求量会因此而加大。

如果拿不准是否要进行升级，总归可以在不干扰现有的MySQL服务器的前提下试用新MySQL服务器一段时间。可以让新、旧服务器同时运行在同一台主机上，也可以把新服务器安装到另外一台机器上。让新服务器运行在另一台机器上的做法比较容易保证新、旧两个服务器不会相互干扰，因为你有更大的自由度去按自己的想法配置它。如果你选择的是让新、旧服务器同时运行在同一台主机上，千万要让新服务器的各种配置参数（比如安装路径、数据目录以及它所监听的网络端口和套接字）不同于旧服务器。这方面的具体情况请参见第11.6节。

如果你决定先试用新MySQL服务器，你很可能想用现有数据库的一份拷贝去试用它。对数据库进行拷贝的具体步骤请参见第13章中的有关内容。

如果你决定对MySQL进行升级，请先阅读MySQL发行版本自带的MySQL Reference Manual

(MySQL参考手册)中的“Change Notes”(新增功能)附录,看其中有没有提到一些需要在升级时注意的特别事项——很可能没有,但无论如何要看一下。这一步非常重要,尤其是在新版本引入了一些与老版本不兼容的操作行为时。

与升级时相比,如果你把MySQL升级到了一个不向后兼容的版本之后又后悔了,再想“降级”到老版本可能就没那么容易了。比如说,如果你把MySQL从4.0.x升级到了4.1.x并已经把数据表全都转换成了4.1格式,它们就与4.0.x版本的MySQL服务器不兼容了,你很难再对它们进行“降级”。

#### 不要害怕试用开发版本

把一个开发版本用做业务目的(比如管理你的商务资料)是不明智的。但仍鼓励大家去试用新版本,而且最好是用日常业务数据的一份拷贝来进行试用。要知道,试用新版本的人越多,对新版本的检验就越彻底,在新版本里发现程序漏洞的几率就越大,这其实是件好事。来自用户团体的程序漏洞报告是帮助MySQL开发工作向前推进的一个重要因素,它们能帮助开发人员发现问题并加以改进。

如果你希望在试用新版本时有一个源源不断的查询命令来源,不妨把现有的MySQL服务器用做镜像机制中的主服务器,把新MySQL服务器用做镜像机制中的从服务器。这样,在主服务器上执行的每一条查询命令都将被发送到从服务器上再执行一遍,使从服务器获得一个连续不断的查询命令流。这也使你有机会了解新服务器在正常的日常工作中到底能有多好的表现。

## 第12章 MySQL安全技术

MySQL系统管理员有责任维护好MySQL数据库的安全性和完整性。在第11章中，我们已经涉及到了一些有关安全技术的话题，如设置初始MySQL root口令的重要性、如何创建用户账户等等。这些话题是作为MySQL的安装和启动过程的一部分而介绍给大家的。在这一章里，将对以下几个与安全技术有关的问题做进一步的讨论：

- 安全技术的重要性以及可能会遭受到的攻击。
- 内部安全性，即在MySQL服务器主机上有登录账户的其他用户对你的威胁及你的防范措施。
- 外部安全性，即通过网络连接到MySQL服务器的客户程序对你的威胁及你的防范措施。

维护数据库内容的安全是MySQL管理员的职责之一，应该只让拥有适当授权的人们去访问数据库里的记录。这又分为内部安全性和外部安全性两方面。内部安全性指的是与能够直接访问MySQL服务器主机的其他用户（即那些在MySQL服务器主机上有登录账户的其他用户）有关的问题。一般来讲，内部安全漏洞大都与MySQL服务器主机的文件系统有关，你需要保护MySQL数据库系统免遭在MySQL服务器主机上有账户的人们的攻击。尤其是数据目录，它应该只能由你用来运行MySQL服务器的那个登录账户拥有和控制。如果做不到这一点，在安全方面的其他努力就可能白费。比如说，经网络连接到MySQL服务器的客户的访问权限是由权限表来控制的，可即使你对权限表里列出的账户进行了正确的设置，如果你把数据目录内容的访问模式设置得过于宽松的话，只要替换掉相应的权限表文件，别人就能轻易地改变你所设置的客户访问控制策略。

外部安全性指的是与从外部网络连接到MySQL服务器的客户程序有关的问题，即保护MySQL服务器免遭那些请求访问数据库内容的网络连接的攻击。应该把MySQL权限表设置成只允许那些能提供出合法的用户名和口令的账户去访问它所管理的数据库。另一种危险是有人在监控着网络并试图捕获在MySQL服务器与用户的客户程序之间传输的信息，如果这让你感到不安的话，就需要把MySQL安装配置成支持使用SSL（Secure Socket Layer，安全套接字层）协议来进行连接。

在这一章里，将对以上这些需要你提高警惕的问题以及如何在内、外两方面防止出现未经授权的访问进行介绍。在讨论中，我们经常要用到一个用来运行MySQL服务器和完成各种与MySQL有关的管理性工作的登录账户，这个账户在文中的用户名和用户组名分别是mysqladm和mysqlgrp，如果你使用的用户名和用户组名与此不同，请在有关的操作示例中改变它们的名字。

### 12.1 内部安全性：防止未经授权的文件系统访问

本节将向大家介绍如何对安装在机器里的MySQL进行安全处理以防止MySQL服务器主机上的未经授权用户对它进行篡改和破坏。本节仅适用于UNIX系统；如果你的MySQL服务器运行在Windows系统上，将假设你对那台主机有着绝对的控制权，而该主机也没有其他的本地用户。



(换句话说,最好不要让其他用户有机会接触这台运行着MySQL服务器的Windows主机,严格限制人们靠近这台机器。)

MySQL的安装过程会创建几个目录,有些目录必须受到与其他目录不同的特殊保护。比如说,除MySQL管理员账户外,没必要让其他人有访问MySQL服务器程序文件的权限。但MySQL客户程序却应该是每位MySQL用户都能执行的——但这些程序应该不允许任何人去修改或者替换。

某些需要受到保护的文件是在安装工作基本结束之后才被创建出来的,它们有些是在安装阶段的后期配置工作中由你本人创建的,有些是由MySQL服务器在启动和运行阶段创建。需要由你本人去创建的文件包括各种选项文件和与SSL有关的文件。由MySQL服务器在运行阶段创建的目录和文件包括数据目录、对应于各数据库的目录、对应于各数据表的文件、日志文件、UNIX套接字文件等等。

很显然,在MySQL服务器管理下的各个数据库应该由它们各自的主人“全权处置”。数据库的主人通常都会认为数据库里的内容是他们的私有财产——这一点是受法律保护的,即使他们不这样想,也应该由他们来决定要把自己的数据库内容公开给谁,决不能因为你没能保护好数据目录而让它们泄露出去。

日志文件都必须得到妥善的保护,它们记载着各种查询命令的文本,假如某人有权对日志文件进行访问,他就能掌握数据库内容的变化。那些记载在日志文件里的GRANT和SET PASSWORD等语句就更是安全重点了,因为这些语句里包含着口令等敏感信息。MySQL会对口令进行加密,但这只发生在使用已设置好的口令去连接MySQL服务器的情况下;而在人们使用GRANT、INSERT或SET PASSWORD等语句去设置口令的过程中,这些查询命令都是以明文形式被记载到某些日志文件里去的。如果攻击者有权读取日志文件的内容,就能轻易地通过用grep程序在日志文件里搜索GRANT或PASSWORD等单词的办法获得这些敏感的信息。

至于UNIX套接字文件等允许各种客户程序访问的文件,应该把客户程序对它们的访问限制在使用而不是控制方面。比如说,如果某位用户有权删除套接字文件,就会对本地主机上的客户构成威胁。

### 12.1.1 如何偷取数据

下面这个简单的例子揭示了安全工作的重要意义,它能让我们明白为什么不应该让其他用户拥有直接访问MySQL数据目录的权限。

MySQL服务器通过mysql数据库里的权限表向我们提供了一整套灵活的权限控制机制。通过在权限表里做出的设置,你可以随心所欲地允许或者拒绝有关用户对数据库的访问,让那些未经授权的用户无法通过网络来访问你的数据。可是,如果MySQL服务器主机上的其他用户能直接访问MySQL数据目录的内容的话,你为数据库设置的网络访问控制策略再好,也只能算做是一次毫无实际意义的练习。如果你知道在MySQL服务器主机上有登录账户的并非仅有你一人,就应该禁止那台机器上的其他登录账户访问数据目录。

很显然,还应该禁止其他服务器主机上的用户直接对数据目录进行写操作,因为他们很可能把你的状态文件或数据表文件覆盖掉。可即便只允许其他服务器主机上的用户直接对数据



目录进行读操作也是危险的。如果某位用户有权直接读取数据表文件，就可以轻易偷走这个文件并让MySQL把它的内容显示出来，如下所示：

1) 在你自己的主机上再安装一个“非法的”MySQL服务器，但它的端口、套接字和数据目录要与“合法的”MySQL服务器不同。

2) 运行mysql\_install\_db脚本对数据目录进行初始化。这将使你能够以MySQL root用户的身份去访问“非法的”MySQL服务器并在其中创建一个test数据库——可以把“偷”来的数据表放在这个数据库里。

3) 进入“合法的”MySQL服务器的数据目录，把你想“偷”的数据表文件拷贝到“非法的”MySQL服务器的数据目录下的test目录里。这个操作只要求你具备对“合法的”数据目录的读权限。

4) 启动“非法的”MySQL服务器。所“偷”来的数据表已经包含在其test数据库里了，可以对它为所欲为。先用SHOW TABLES FROM test语句看着自己都“偷”来了哪些数据表，再用SELECT \*语句去看看它们的内容。就这么简单！

5) 如果你想玩大一点儿，还可以在“非法的”MySQL服务器上创建一个允许任何人从任何主机都能连接到这个test数据库的匿名账户。这将使全世界的人都能看到你“偷”来的数据表。

好好想想这种情况，然后再反过来想一想。你想让别人对你这样做吗？当然不想，那就用下面介绍的各种方法来保护你自己。

### 12.1.2 保护你的MySQL安装程序

在安装好MySQL软件之后，请按以下步骤对安装过程所创建的目录和文件的属主和访问模式进行设置。在下面的示例中，将以mysqladm和mysqlgrp作为有关的目录和文件的用户名和用户组名。我们使用了一个标准的目录布局，即把MySQL软件各组成部分都统一安装在一个基本安装路径之下而不是让它们散布文件系统的各个地方。所使用的基本安装路径是/usr/local/mysql目录，并把MySQL数据目录的路径名设定为/usr/local/mysql/data。在介绍完标准布局下的安全处理步骤之后，再对一些不标准的安装布局下的安全处理办法进行介绍。如果读者的系统布局与这里描述的不太一致，请根据有关步骤的基本原理加以变通。别忘了把各有关示例中的用户名和路径名改成你在自己的系统上使用的名字。如果你运行着多个MySQL服务器，就需要按以下步骤对每一个MySQL服务器分别进行一次安全处理。

首先，用ls -la命令看看MySQL数据目录里都有哪些不安全的文件或目录。我们要找的是那些“group”或“other”权限被打开的文件和目录。下面是某个不安全的数据目录的文件清单，可以在其中看到一些数据库目录：

```
% ls -la /usr/local/mysql/data
total 10148
drwxrwxr-x  11 mysqladm wheel      1024 May  8 12:20 .
drwxr-xr-x  22 root        wheel      512 May  8 13:31 ..
drwx-----  2 mysqladm mysqlgrp    512 Apr 16 15:57 menagerie
drwxrwxr-x  2 mysqladm wheel      512 Jun 25  1998 mysql
drwx-----  7 mysqladm mysqlgrp   1024 May  7 10:45 sampdb
drwxrwxr-x  2 mysqladm wheel     1536 Jun 25  1998 test
drwx-----  2 mysqladm mysqlgrp   1024 May  8 18:43 tmp
```

在上面这份清单里，有几个数据库目录的访问模式是适当的：“drwx-----”只允许属主进行读、写和执行访问，但不允许别人做任何访问。但另外一些目录的访问模式却设置得过于宽松了：“drwxrwxr-x”允许任何人甚至不是mysqlgrp用户组的成员进行读和执行访问。这是一个已经存在了很长时间的数据库系统，它是因为从一个早期的版本不断升级到新版本而变成现在这个样子的。那些有问题的访问模式都是老版本的MySQL服务器设置的——老版本的MySQL服务器在设置访问权限时不像新版本这么严密。（请注意，访问模式被设置得较为严密的数据库目录如menagerie、sampdb和tmp等都有比较近的创建日期。）最新版本的MySQL服务器现在都把自己所创建的数据库目录上的访问模式设置为只允许自己在其名下运行的那个账户进行访问。

接下来，再用ls -l命令来检查一下这个MySQL软件的基本安装路径。比如说，你可能会看到如下所示的结果：

```
% ls -la /usr/local/mysql
total 44
drwxrwxr-x 13 mysqladm mysqlgrp 1024 May 7 10:45 .
drwxr-xr-x 24 root wheel 1024 May 1 12:54 ..
drwxr-xr-x 2 mysqladm mysqlgrp 1024 Jul 16 20:58 bin
drwxrwxr-x 12 mysqladm wheel 1024 May 8 12:20 data
drwxr-xr-x 3 mysqladm mysqlgrp 512 May 7 10:45 include
drwxr-xr-x 2 mysqladm mysqlgrp 512 May 7 10:45 info
drwxr-xr-x 3 mysqladm mysqlgrp 512 May 7 10:45 lib
drwxr--r-x 2 mysqladm mysqlgrp 512 Jul 16 20:58 libexec
drwxr-xr-x 3 mysqladm mysqlgrp 512 May 7 10:45 man
drwxr-xr-x 6 mysqladm mysqlgrp 1024 May 7 10:45 mysql-test
drwxr-xr-x 3 mysqladm mysqlgrp 512 May 7 10:45 share
drwxr-xr-x 7 mysqladm mysqlgrp 1024 May 7 10:45 sql-bench
```

data目录的访问模式和属主需要修改，理由和前面提到的一样。另一处需要修改的地方是把mysqld服务器程序所在的libexec目录的访问模式设置得再严密一些：除MySQL管理员以外的任何人都需要访问MySQL服务器程序，所以这个目录的访问模式应该被设置成只允许mysqladm用户访问。

我们将按以下步骤来消除上面提到的这些隐患。这里的基本原则是让所有东西（除那些允许合法用户去合法访问的部分外）都只允许mysqladm用户去访问：

1) 如果MySQL服务器正在运行，则关停之：

```
% mysqladmin -p -u root shutdown
```

2) 用以下命令把整个MySQL安装程序的属主名和用户组名设置为MySQL管理员账户的用户名和用户组名（必须以系统的root用户身份执行这些命令）：

```
# chown -R mysqladm.mysqlgrp /usr/local/mysql
```

另一种比较常见的做法是把数据目录以外的一切都设置为由root用户拥有，如下所示：

```
# chown -R root.mysqlgrp /usr/local/mysql
```

```
# chown -R mysqladm.mysqlgrp /usr/local/mysql/data
```

如果把整个MySQL安装程序的属主设置为root，下面的大部分操作就必须以root用户身份来

执行；否则，可以用mysqladm用户身份来执行。

3) 对于允许客户程序去访问的基本安装目录及各有关目录，需要把它们的访问模式修改成允许mysqladm用户进行各种访问、但只允许其他人进行读和执行访问。它们可能已经被设置好了，但如果不是这样，请改变之。比如说，基本安装目录可以用下面两条命令中的任何一条来设置：

```
% chmod 755 /usr/local/mysql
% chmod u=rwx,go=rx /usr/local/mysql
```

同样，用来存放客户程序的bin目录可以用下面两条命令中的任何一条来设置：

```
% chmod 755 /usr/local/mysql/bin
% chmod u=rwx,go=rx /usr/local/mysql/bin
```

客户程序不需要访问的目录要设置为只允许mysqladm访问，用来存放MySQL服务器程序文件的libexec目录就是一个例子，可以用下面两条命令中的任何一条来设置它的访问模式：

```
% chmod 700 /usr/local/mysql/libexec
% chmod u=rwx,go-rwx /usr/local/mysql/libexec
```

4) 把数据目录以及其中的所有文件的目录的访问模式改变为只允许mysqladm用户访问。这样，除用来运行MySQL服务器的mysqladm账户外，任何其他账户就都不能直接访问数据目录里的东西了。要用下面这条命令来做这件事：

```
% chmod -R go-rwx /usr/local/mysql/data
```

完成上述步骤之后，MySQL基本安装目录的属主和访问模式将如下所示：

```
% ls -la /usr/local/mysql
total 44
drwxr-xr-x 13 mysqladm mysqlgrp 1024 May 7 10:45 .
drwxr-xr-x 24 root wheel 1024 May 1 12:54 ..
drwxr-xr-x 2 mysqladm mysqlgrp 1024 Jul 16 20:58 bin
drwx----- 12 mysqladm mysqlgrp 1024 May 8 12:20 data
drwxr-xr-x 3 mysqladm mysqlgrp 512 May 7 10:45 include
drwxr-xr-x 2 mysqladm mysqlgrp 512 May 7 10:45 info
drwxr-xr-x 3 mysqladm mysqlgrp 512 May 7 10:45 lib
drwx----- 2 mysqladm mysqlgrp 512 Jul 16 20:58 libexec
drwxr-xr-x 3 mysqladm mysqlgrp 512 May 7 10:45 man
drwxr-xr-x 6 mysqladm mysqlgrp 1024 May 7 10:45 mysql-test
drwxr-xr-x 3 mysqladm mysqlgrp 512 May 7 10:45 share
drwxr-xr-x 7 mysqladm mysqlgrp 1024 May 7 10:45 sql-bench
```

如上所示，所有东西现在都由mysqlgrp用户组里的mysqladm用户拥有了。（清单中的第二项“..”代表着目录/usr/local/mysql的父目录。这个父目录由root用户拥有，也只允许root用户修改。应该这样设置——你肯定不想让未经授权的用户把MySQL基本安装目录的父目录弄得一团糟。）

基本安装目录下的数据目录有着更严格的访问模式：

```
% ls -la /usr/local/mysql/data
total 10148
```

```

drwx----- 11 mysqladm mysqlgrp 1024 May 8 12:20 .
drwxr-xr-x 22 mysqladm mysqlgrp 512 May 8 13:31 ..
drwx----- 2 mysqladm mysqlgrp 512 Apr 16 15:57 menagerie
drwx----- 2 mysqladm mysqlgrp 512 Jun 25 1998 mysql
drwx----- 7 mysqladm mysqlgrp 1024 May 7 10:45 sampdb
drwx----- 2 mysqladm mysqlgrp 1536 Jun 25 1998 test
drwx----- 2 mysqladm mysqlgrp 1024 May 8 18:43 tmp

```

这个清单里的第二项“..”代表着数据目录的父目录，也就是MySQL基本安装目录。

某些特定的文件是这种“数据目录中的一切都只允许mysqladm用户访问”策略的例外。比如说，如果在数据目录里创建了一个my.cnf选项文件，那么，如果打算把一些客户选项放在这个文件里，就必须适当放松对数据目录的访问控制，否则客户程序将无法读取这个文件。如果打算把UNIX套接字文件也放在数据目录里，也会遇到同样的问题。下面这条命令能够让客户程序在不具备读权限的前提下使用数据目录中的选项文件或套接字文件：

```
% chmod go+x /usr/local/mysql/data
```

如前所述，上述过程只适用于与MySQL有关的所有文件都放在同一个基本安装目录下的情况。如果不是这样，就得依次对每一个与MySQL有关的目录进行上述处理。比如说，如果MySQL数据目录位于/var/mysql/data而不是位于/usr/local/mysql目录下，就需要用下面两条命令来改变它的属主：

```
# chown -R mysqladm.mysqlgrp /usr/local/mysql
# chown -R mysqladm.mysqlgrp /var/mysql/data
```

或者，假设你在MySQL基本安装目录下创建了一个innodb目录来存放所有与InnoDB有关的文件。在默认情况下，这些文件将被放到数据目录里去。如果要把它们放到innodb目录里，就需要把它设置成与数据目录的访问模式相同。这一原则还适用于对某些应该放在数据目录里的文件（比如日志文件）进行了重新安置的情况。

如果基本安装目录下的某些目录实际上是一些指向其他地方的符号链接，事情就会更复杂。如果你使用的chown命令不能跟踪符号链接，就需要由你来找出符号链接的目标位置并对它的属主做相应的修改。可以利用find命令来完成这一工作，如下所示：

```
# find /usr/local/mysql -follow -print | xargs chown mysqladm.mysqlgrp
```

别忘了对目标目录的访问模式做相应的修改。如果数据目录里有一个符号链接而chmod命令不能跟踪符号链接，可以使用下面这样的命令：

```
% find /usr/local/mysql/data -follow -print | xargs chmod go-rwx
```

注意，如果MySQL服务器和客户程序被安装到了某个通用性的系统目录里并与一些非MySQL程序混杂在一起（比如MySQL服务器被安装在/usr/sbin目录里、客户程序被安装在/usr/bin目录里的时候），上面的某些步骤就不一定适用。此时，应该把各有关MySQL程序的属主和访问模式设置成与那些目录里的其他程序文件一样。

从现在起，因为数据目录内容的属主和访问模式被设置成只允许mysqladm用户访问，所以必须总是以mysqladm用户身份去运行MySQL服务器才能成功。为确保这一点，可以在



/etc/my.cnf文件或数据目录中的my.cnf文件里的[mysqld]选项组里加上一条如下所示的user设定指令：

```
[mysqld]
user=mysqladm
```

这样，不管你是以root用户身份登录还是以mysqladm用户身份登录，所启动的MySQL都将以mysqladm用户身份运行。把MySQL服务器运行在特定登录账户名下的具体做法见第11章中的有关内容。

对机器里安装的MySQL进行安全处理后，就可以重新启动MySQL服务器了。

#### 1. 保护套接字文件

对于以localhost为主机名的客户连接，MySQL服务器将使用UNIX域套接字文件。为了让各个客户程序都能使用这个套接字文件，它通常是允许所有用户访问的。但要注意的是，千万不要把它放到一个其他用户在其上拥有删除权限的目录里去。比如说，套接字文件被创建在/tmp目录里是很常见的现象，但某些UNIX系统的/tmp目录的访问模式允许普通用户在其中删除不属于自己的文件——这意味着任何用户都能删除套接字文件，这样，在MySQL服务器重新启动并重新创建一个套接字文件之前，客户程序将无法以localhost为主机名去连接MySQL服务器。因此，最好能对/tmp目录的“粘滞位”进行置位，这样，哪怕任何用户都能在这个目录里创建文件，用户也只能删除他们自己的文件。

某些安装会把套接字文件放在数据目录里，但这又会导致这样一个问题：如果你把数据目录设置成只允许mysqladm用户访问而你却没有以root或mysqladm用户身份去运行MySQL服务器，客户程序就不能访问套接字文件。此时，为了让客户程序能够“看”到套接字文件，办法之一是把数据目录的访问控制稍微放松一点儿：

```
% chmod go+x /usr/local/mysql/data
```

另一个办法是让MySQL服务器把套接字文件创建在另一个地点——可以在某个全局级选项文件里为套接字文件另外指定一个创建地点，也可以重新编译MySQL软件的源代码并在编译配置阶段为套接字文件另外指定一个不同的默认创建地点。如果你决定使用选项文件，一定要为MySQL服务器和客户程序都设定这个位置。比如说，如果要把套接字文件放在基本安装目录里，就需要在选项文件里做出如下所示的设置：

```
[mysqld]
socket=/usr/local/mysql/mysql.sock
[client]
socket=/usr/local/mysql/mysql.sock
```

因为使用选项文件的办法对于那些不读取选项文件的客户程序不起作用（MySQL自带的标准客户程序都会去读取选项文件，但某些第三程序不一定会去读取选项文件），所以更圆满的解决方案是重新编译MySQL源代码，但这种做法的工作量会稍大一些。经过重新编译之后，新的套接字文件创建地点将成为MySQL C客户程序开发库的默认值；这样，不管是否会去读取选项文件，凡是使用MySQL C客户程序开发库编写出来的程序就都将以新的套接字文件创建地点作为其默认值了。



## 2. 保护选项文件

选项文件是一个潜在的安全漏洞，因为其中的选项往往会包含有一些敏感的信息。作为一条基本原则，如果选项文件里包含有MySQL账户名或口令等敏感信息，就不应该让它对所有用户都可读。一般说来，选项文件/etc/my.cnf总是对所有用户可读的，因为它的内容往往是适用于各种客户程序的全局级客户选项（这意味着不应该把仅供MySQL服务器使用的选项，比如镜像机制的口令，放在这个文件里）。至于仅供每位用户专用的选项文件.my.cnf，应该把它们分别放到各有关用户的登录目录里，即只让这个选项文件由给定用户拥有和可读。根据其具体用途，MySQL数据目录里的选项文件既可以对所有用户可读，也可以只由mysqladm用户拥有和可读。

为确保仅供每位用户专用的.my.cnf选项文件的访问模式和属主都设置得当，办法之一是用一个程序把它们依次查找出来并纠正可能存在的问题；下面这个名为chk\_mysql\_opt\_files.pl的Perl脚本就是做这项工作的：

```
#!/usr/bin/perl -w
# chk_mysql_opt_files.pl - check user-specific .my.cnf files and make sure
# the ownership and mode is correct. Each file should be owned by the
# user in whose home directory the file is found. The mode should
# have the "group" and "other" permissions turned off.

# This script must be run as root. Execute it with your password file as
# input. If you have an /etc/passwd file, run it like this:
# chk_mysql_opt_file.pl /etc/passwd
# For Mac OS X, use the netinfo database:
# nidump passwd . | chk_mysql_opt_file.pl

use strict;
while (<>)
{
    my ($uid, $home) = (split (/:/, $_))[2,5];
    my $cnf_file = "$home/.my.cnf";
    next unless -f $cnf_file;          # is there a .my.cnf file?
    if ((stat ($cnf_file))[4] != $uid) # test ownership
    {
        warn "Changing ownership of $cnf_file to $uid\n";
        chown ($uid, (stat ($cnf_file))[5], $cnf_file);
    }
    my $mode = (stat ($cnf_file))[2];
    if ($mode & 077)                    # test group/other access bits
    {
        warn sprintf ("Changing mode of %s from %o to %o\n",
                        $cnf_file, $mode, $mode & ~077);
        chmod ($mode & ~077, $cnf_file);
    }
}
exit (0);
```

可以在sampdb发行版本的admin目录中找到chk\_mysql\_opt\_files.pl脚本。必须以root用户身份去执行这个脚本，因为它要去改变由其他用户拥有的文件的访问模式和属主。如果你想让这个脚本定期地自动执行，可以把它安排为一个会在每天晚上以root身份运行的cron作业。

每一个MySQL程序都会到被编译在MySQL软件里的数据目录里去寻找名为my.cnf的标准选项文件。但如果把这个数据目录设置成只允许mysqladm用户访问，客户程序将不能读取该文件。这意味着，如果打算把客户参数放在这个文件里，就必须放松对数据目录的访问控制——需要在数据目录的访问模式里打开execute（执行）权限，还要让my.cnf文件是对所有用户可读的，如下所示：

```
% chmod go+x /usr/local/mysql/data
% chmod +r /usr/local/mysql/data/my.cnf
```

不过，如果你这样做了，再把仅供MySQL服务器使用的选项（比如镜像机制的口令）放在这个文件里就不安全了。

## 12.2 外部安全性：防止未经授权的网络访问

MySQL的安全系统是很灵活的，它允许你以多种不同的方式去设置各位用户的访问权限。比较常见的做法是由你发出GRANT和REVOKE语句，再由它们去修改各有关权限表里的客户权限信息。不过，如果你使用的是一个尚不支持这些语句的MySQL老版本（MySQL 3.22.11之前的版本）或者发现你为用户设置的权限达不到预期的效果，多知道些关于MySQL权限表的结构、功能和用法的知识还是很有帮助的。只有掌握了这些知识，你才能通过直接修改权限表的办法去增加、删除或修改用户权限，才能对权限设置方面的问题进行追踪和纠正。

这里假设大家已经阅读过第11.3节并熟悉GRANT和REVOKE语句的工作原理。GRANT和REVOKE语句使我们能够方便地对MySQL用户账户及其权限进行设置，但这两条语句只是一个前端，真正的动作发生在MySQL数据库系统的权限表里。（事实上，直接修改权限表也能获得与发出GRANT语句同样的效果，具体做法见第12.2.5节。）

### 12.2.1 MySQL权限表的结构和内容

经网络连接到MySQL服务器的客户程序在MySQL数据库上的访问权限由权限表的内容控制。这些权限表都存放在mysql数据库里，它们会在MySQL被首次安装到某台主机时被初始化（具体过程见本书的附录A）。这些权限表的名字是user、db、tables\_priv、columns\_priv和host，它们的具体用途如下所述：

- user权限表：其内容是允许连接MySQL服务器的各个用户的账户信息，如用户名、口令、全局级（超级用户）权限（如果有的话）等等。在user权限表里激活的权限都是适用于所有数据库的全局级权限，这一点请大家务必牢记。比如说，如果你在user权限表的某个记录项里激活了DELETE权限，该记录项所对应的账户就能从任何一个数据库里的任何一个数据表里删除数据记录。因此，是否要在user权限表里激活某项权限一定要三思

而后行。

因为在user权限表里激活的权限都具有超级用户效力，所以最好不要在这个权限表的记录项里激活任何权限——应该在其他几个对权限有着更严格控制的权限表里为用户设定更具体的操作权限。但下面两种情况属于这个原则的例外：

- 超级用户（比如root和其他管理员账户）必须具备全局级权限才能控制MySQL服务器的运转。但这类账户在数量上往往很少。
- 某些特定的全局级权限（比如创建临时数据表、锁定数据表或者发出SHOW DATABASES语句所需要的几种权限）通常可以安全地授予给用户。绝大多数MySQL数据库系统都会把这些权限授予各位用户；可如果你的MySQL数据库系统关系重大，需要更为严格的安全控制，就不要在user权限表里激活这些权限。

user权限表还有一些用来存放SSL选项（用来建立SSL连接）和资源管理选项（用来防止某给定账户把MySQL服务器主机上的系统资源消耗殆尽）的数据列。

- db权限表：其内容是各个账户在各个数据库上的操作权限。在这个权限表里授予的权限将作用于给定数据库里的所有数据表。
- tables\_priv权限表：用来设定数据表级的操作权限。在这个权限表里授予的权限将作用于给定数据表里的所有数据列。
- columns\_priv权限表：用来设定数据列级的操作权限。在这个权限表里授予的权限将作用于给定数据表里的给定数据列。
- host权限表：配合db权限表对给定主机上的数据库级操作权限做更细致的控制。这个权限表不受GRANT和REVOKE语句的影响，你很可能永远都不会用到它。

所有的权限表都由两大类别的数据列构成：一类是用来表明权限作用范围的数据列，即有关权限会在何时何地发挥作用；另一类是用来给出权限（名称）的数据列，这些数据列中的权限又可以被进一步划分为管理性操作权限和数据库/数据表操作权限。我们把各个权限表中的数据列按上述办法分门别类地列在表12-1、12-2和12-3里。此外，user权限表还有几个用来存放SSL选项和资源管理选项的数据列；但这几个数据列只出现在user权限表里，因为它们的作用范围是系统全局。有些权限表还有几个杂项数据列，但因为它们都与账户管理工作无关，所以这里就不再把它们列出来了。

表12-1 权限表中的权限作用范围数据列

权限作用范围数据列				
user权限表	db权限表	tables_priv权限表	columns_priv权限表	host权限表
Host	Host	Host	Host	Host
User	User	User	User	
Password	Db	Db	Db	Db
		Table_name	Table_name	
			Column_name	

表12-2 权限表中的权限数据列

管理权限数据列		
user权限表	db权限表	host权限表
Create_tmp_table_priv	Create_tmp_table_priv	Create_tmp_table_priv
Execute_priv		
File_priv		
Grant_priv	Grant_priv	Grant_priv
Lock_tables_priv	Lock_tables_priv	Lock_tables_priv
Process_priv		
Reload_priv		
Repl_client_priv		
Repl_slave_priv		
Show_db_priv		
Shutdown_priv		
Super_priv		
数据库/数据表权限数据列		
user权限表	db权限表	host权限表
Alter_priv	Alter_priv	Alter_priv
Create_priv	Create_priv	Create_priv
Delete_priv	Delete_priv	Delete_priv
Drop_priv	Drop_priv	Drop_priv
Index_priv	Index_priv	Index_priv
Insert_priv	Insert_priv	Insert_priv
References_priv	References_priv	References_priv
Select_priv	Select_priv	Select_priv
Update_priv	Update_priv	Update_priv
tables_priv权限表	columns_priv权限表	
Table_priv	Column_priv	

表12-3 权限表中的SSL和资源管理数据列（仅user权限表有）

SSL数据列	资源管理数据列
ssl_type	max_connections
ssl_cipher	max_questions
x509_issuer	max_updates
x509_subject	

tables\_priv和columns\_priv权限表用来设置数据表级和数据列级权限。不过，因为MySQL没有提供数据记录级权限，所以不存在rows\_priv权限表之类的东西——比如说，无法限制某位用户只能访问某数据表里特定的某几个数据行；如果需要这种能力，就只能自行编写相应的应用程序（可以用GET-LOCK()函数来实现数据记录级访问控制，详见本书的附录C）。

### 权限表的结构改变了怎么办

权限表的结构时不时地会发生一些改变——当你在MySQL服务器上查看mysql数据库中的数据表时，很可能会发现有些数据表或者数据列并不存在：

- tables\_priv和columns\_priv权限表最早出现于MySQL 3.22.11版本（与GRANT语句同时）。如果你使用的MySQL软件版本较早，你的mysql数据库里就可能只有user、db和host权限表。
- user权限表里的SSL数据列最早出现于MySQL 4.0.0版本。
- user权限表里的资源管理数据列最早出现于MySQL 4.0.2版本。
- user权限表里的以下权限数据列最早出现于MySQL 4.0.2版本：Create\_tmp\_table\_priv、Execute\_priv、Lock\_tables\_priv、Repl\_client\_priv、Repl\_slave\_priv、Show\_db\_priv和Super\_priv。db和host权限表里的Creat\_tmp\_table\_priv和Lock\_tables\_priv权限最早出现于MySQL 4.0.4版本。

如果你的权限表结构与上面介绍的不一样，可以用mysql\_fix\_privilege\_tables脚本对它们进行升级。这个脚本需要以MySQL数据库系统的root用户身份去连接本地服务器，需要像下面这样给出一个适当的口令：

```
% mysql_fix_privilege_tables root-password
```

如果你是从一个老版本升级到MySQL 4.0.2或更高版本的，mysql\_fix\_privilege\_tables脚本会给user权限表增加必要的权限数据列并按以下规则对现有user权限表中的非匿名账户进行初始化：

- 把Show\_db\_priv设置为当前的Select\_priv值。
- 把Execute\_priv和Super\_priv设置为当前的Process\_priv值。把Repl\_client\_priv和Repl\_slave\_priv设置为当前的File\_priv值。
- 把Create\_tmp\_table\_priv和Lock\_tables\_priv设置为'Y'。

如有可能，最好跳过4.0.2和4.0.3版本直接升级到4.0.4或更高的版本。这是因为4.0.2和4.0.3版本中的db权限表没有Creat\_tmp\_table\_priv和Lock\_tables\_priv权限，所以你将无法在数据库级授予这两种权限，而这会给MySQL数据库管理工作带来一些困难。

#### 1. 权限表中的权限作用范围数据列

当某给定账户试图执行一项给定操作时，MySQL将使用权限表中的权限作用范围数据列来判断这个操作将作用在哪些数据行上。每个权限表记录项都包含有Host和用户数据列，它们的作用是表明该记录项适用于哪个用户从哪个主机建立的连接。比如说，如果某个user权限表记录项的Host和用户数据列分别是localhost和bill，就表明该记录项适用于用户bill而不是用户betty从本地主机建立的连接。（host权限表是个例外，稍后介绍。）其他权限表比user权限表多出了一个或几个权限作用范围数据列：db权限表多出了一个用来表明该记录项适用于哪一个数据库的Db数据列，tables\_priv权限表比db权限表多出了一个用来表明该记录项适用于哪一个数据库里的哪一个数据表的Table\_name数据列，columns\_priv权限表又比tables\_priv权限表再多出了一个用来



表明该记录项适用于哪个数据表里的哪个数据列的Columns\_name数据列。

## 2. 权限表中的权限名称数据列

权限表还包含有一些权限数据列。这些数据列的作用是表明与权限作用范围数据列值相匹配的用户都拥有哪些权限。下面是MySQL目前支持的各种权限的清单，这里把它们分为管理权限和数据库/数据表访问权限两部分来介绍。在清单中，各有关权限按它们用在GRANT语句中的名字排列。绝大多数的权限名都与user、db和host权限表中的权限数据列名非常相似。比如说，SELECT权限就对应于Select\_priv数据列。

### (1) 管理权限

下列权限分别对应着用来控制MySQL服务器运行情况的管理性操作或用户的权限授予能力：

- **CREAT TEMPORARY TABLES** 允许你用CREAT TEMPORARY TABLE语句创建临时数据表。
  - **EXECUTE** 允许你执行“存储过程”。这个权限目前并未实现，它将在今后的MySQL版本里（目前的计划是在MySQL 5中）随“存储过程”机制的实现而生效。
  - **FILE** 允许你通过MySQL服务器去读、写MySQL服务器主机上的文件。为了把这个权限控制在一定范围内，MySQL服务器在读、写服务器主机上的文件时要遵守以下规则：
    - 只能通过MySQL服务器去访问对任何用户可读的文件，即那些不受任何保护的文件。
    - 如果想通过MySQL服务器写入一个文件，该文件必须尚不存在。这是为了防止你通过MySQL服务器写入的文件覆盖掉系统中的重要文件，比如/etc/passwd文件或者由别人拥有的数据库里的数据库文件（如果没有这条限制，你就能整个地替换掉mysql数据库里的权限表的内容）。
- 尽管有这些预防措施，也不应该在没有任何充足理由的情况下把这个权限授予给普通用户，那会非常危险——详细讨论见第12.2.4节。如果你必须向别人授予FILE权限，就千万不要以UNIX系统的root用户身份去运行MySQL服务器，因为root用户能够在文件系统中的任何地方创建新文件。应该用一个普通的登录账户来运行MySQL服务器，这样，MySQL服务器就只能在该账户有权访问的目录里创建文件了。
- **GRANT OPTION** 允许你把自己的权限（包括GRANT OPTION权限在内）授予其他用户。
  - **LOCK TABLES** 允许你明确地发出LOCK TABLES语句来锁定数据表。这个权限的作用范围仅限于你还拥有其SELECT权限的数据表，但允许你设置读操作锁和写操作锁，而不仅仅是读操作锁。MySQL服务器在处理由你发出的查询命令时隐含地替你申请的操作锁与这个权限无关——MySQL服务器将自动设置和释放那些操作锁而不管你有没有LOCK TABLE权限。
  - **PROCESS** MySQL服务器是多线程的，每条客户连接分别由一个单独的线程进行处理。这些线程可以被看做是在MySQL服务器内部运行的一些进程。在MySQL 4.0.2之前的版本里，PROCESS权限允许你使用SHOW PROCESSLIST语句或mysqladmin processlist命令去查看当前正在执行的线程，它还允许你使用KILL语句或mysqladmin kill命令去终止线程。（即便没有PROCESS权限，你也总是能查看或终止自己的线程。但这个权限将使你具备全局级的能力，即允许你查看或终止任何线程——哪怕它们是属于其他用户的。）PROCESS

权限还允许你使用mysqladmin debug命令，并能在连接MySQL服务器时覆盖max\_connections设定值，使你即使在所有的正常连接槽位都被其他用户占用的情况下也能利用MySQL服务器为管理员保留的连接槽位连接上它。

在MySQL 4.0.2及以后的版本里，PROCESS权限只剩下了查看线程的能力；终止线程（以及使用mysqladmin debug命令和使用保留连接槽位）的能力现在由SUPER权限控制。

- **RELOAD** 允许你进行好几种MySQL服务器管理操作。这个权限将使你具备发出FLUSH和RESET等语句的能力。它还允许你执行以下mysqladmin命令：reload、refresh、flush-hosts、flush-logs、flush-privileges、flush-status、flush-tables和flush-threads。
- **REPLICATION CLIENT** 允许你查询镜像机制中的主服务器和从服务器的位置。
- **REPLICATION SLAVE** 允许某个客户连接到镜像机制中的主服务器并请求它发送二进制变更日志。应该把这个权限授予给镜像机制中的从服务器用来连接主服务器的那个账户；在MySQL 4.0.2之前的版本里，镜像机制中的从服务器是用FILE权限建立的。
- **SHOW DATABASES** 控制着用户发出SHOW DATABASES语句的能力。
- **SUPER** 允许你终止MySQL服务器线程、使用mysqladmin debug命令、使用保留连接槽位（参见对PROCESS权限的说明）。这个权限还允许你使用CHANGE MASTER、PURGE MASTER LOGS以及用来修改全局级MySQL服务器变量和全局级事务处理隔离级别的SET语句。SUPER权限还允许你根据存放在DES密钥文件里的密钥进行DES解密工作。

## （2）数据库和数据表权限

下列权限分别对应着数据库和数据表上的操作：

- **ALTER** 允许你使用ALTER TABLE语句；但根据你打算在某数据表上进行的具体操作，可能还需要具备其他一些权限才行。
- **CREATE** 允许你创建数据库和数据表。这个权限不允许你在数据表上创建索引，除非它们是在CREATE TABLE语句中已经声明过的。
- **DELETE** 允许你从数据表中删除现有的记录。
- **DROP** 允许你丢弃数据库和数据表，但这个权限不允许你丢弃索引。
- **INDEX** 允许你从数据表里创建或丢弃索引。
- **INSERT** 允许你把新的记录插入到数据表里去。
- **REFERENCES** 目前未使用。它最终可能用来规定谁有权设置外键约束条件。
- **SELECT** 允许你使用SELECT语句从数据表中检索数据。但诸如SELECT NOW()或SELECT 4/2之类只进行表达式计算而不涉及任何数据表的SELECT语句不受这个权限的限制。
- **UPDATE** 允许你修改数据表中现有的记录。

有些操作需要你具备多个权限才能进行。比如说，REPLACE操作相当于一个DELETE操作加一个INSERT操作，所以需要同时具备DELETE和INSERT两个权限。

## （3）权限数据列的存储结构

在user、db和host权限表里，每种权限都分别对应着一个数据列。这些权限数据列都被声明为ENUM('N', 'Y')类型且默认值全部是'N'（意思是“不具备这项权限”）。下面是对Select\_priv数据列的定义：

```
Select_priv ENUM('N','Y') NOT NULL DEFAULT 'N'
```

tables\_priv和columns\_priv权限表中的权限是用一个SET类型来表示的，SET类型允许你把多种权限保存在同一个数据列里。下面是tables\_priv权限表中的Table\_priv数据列的定义：

```
SET('Select','Insert','Update','Delete','Create','Drop',
    'Grant','References','Index','Alter')
```

columns\_priv权限表中的Column\_priv数据列定义如下：

```
SET('Select','Insert','Update','References')
```

注意，数据列上的权限要比数据表上的权限少一些，这是因为对数据列有意义的操作在数量上相对较少。比如说，可以从某个数据表里删除它的某个数据行，但无法只删除该数据行的某一个数据列。

tables\_priv和columns\_priv权限表比另外三个权限表出现得晚一些，所以它们采用了更高效的SET类型来存放有关的权限。（未来的user、db和host权限表很可能也会使用SET类型的数据列来存放有关的权限。）

user权限表还有几个其他权限表所没有的管理权限数据列，比如File\_priv、Process\_priv、Reload\_priv和Shutdown\_priv等。这些权限之所以仅出现在user权限里，是因为它们都是与任何一个特定的数据库或数据表都没有关系的全局级权限。比如说，关停MySQL服务器就是关停整个MySQL服务器，允许或不允许某位用户根据他当前使用的是哪一个数据库去关停MySQL服务器是没有意义的。

### 3. 权限表中与SSL有关的数据列

user权限表中还有几个用来验证SSL加密连接的数据列，其中最主要的是用来表明是否需要使用加密连接以及使用哪一种安全连接的ssl\_type数据列。ssl\_type数据列被声明为一个有以下四种可取值的ENUM类型：

- 'NONE'——表示不需要加密连接。这是它的默认值；如果在创建某个账户时没有给出任何REQUIRE子句或者给出的是REQUIRE NONE子句，对应于该账户的ssl\_type数据列就将取值为'NONE'。
- 'ANY'——表示需要加密连接，但可以是任何一种加密连接；这相当于一项“基本”要求。当在GRANT语句里使用了REQUIRE SSL子句时，相应的ssl\_type数据列就会被设置为'ANY'。
- 'X509'——表示需要加密连接并要求客户必须提供一份有效的X509证书，但这份证书的具体内容无关紧要。如果使用了REQUIRE X509子句，相应的ssl\_type数据列就会被设置为'X509'。
- 'SPECIFIED'——表示加密连接必须满足一定的要求。只要在REQUIRE子句里对ISSUER、SUBJECT或CIPHER的值进行过设置，相应的ssl\_type数据列就会被设置为'SPECIFIED'。

如果ssl\_type数据列的取值不是'SPECIFIED'，MySQL在对客户连接进行身份验证时就将忽略user权限表里与SSL有关的其他数据列的值；只有当ssl\_type数据列的取值是'SPECIFIED'的时

候,MySQL服务器才会去检查user权限表里与SSL有关的其他数据列——如果它们当中有非空值,由客户提供的相关信息就必须与之匹配。user权限表中与SSL有关的其他数据列是:

- `ssl_cipher` 若这个数据列不为空,客户在连接时使用的加密算法就必须与这个值相符。
- `x509_issuer` 若这个数据列不为空,客户提供的X509证书中的ISSUER(签发者)就必须与这个值相符。
- `x509_subject` 若这个数据列不为空,客户提供的X509证书中的SUBJECT(主题)就必须与这个值相符。

user权限表里的`ssl_cipher`、`x509_issuer`和`x509_subject`数据列都是BLOB类型。

#### 4. 权限表中的资源管理数据列

user权限表中的以下数据列允许你对任意给定MySQL账户在服务器主机上的系统资源占用量做出限制:

- `max_connections` 该账户每小时可以连接MySQL服务器的次数。
- `max_questions` 该账户每小时可以发出的查询命令数。
- `max_updates` 该账户每小时可以发出的数据修改类查询命令数。

如果以上数据列中的值是零,则表示“没有限制”。当MySQL服务器重新启动时,这几个计数器将全部从零开始重新计数;重新加载权限表或者发出`FLUSH USER_RESOURCES`语句也有同样的效果。

### 12.2.2 MySQL服务器如何对客户进行访问控制

MySQL的客户访问控制机制分为两个阶段。第一阶段发生在你试图连接MySQL服务器的时候——MySQL服务器将到user权限表里寻找与你使用的主机、用户名、口令相匹配的权限记录项。如果没找到匹配,就不能连接。如果不仅找到了匹配,还(如果你的user权限表版本够高的话)找到了SSL或资源管理数据列,MySQL服务器还会检查以下几个数据列:

- 如果已经超过了`max_connections`数据列所设定的每小时最大连接次数,这次连接将被拒绝。
- 如果user权限表记录项表明需要加密连接,MySQL服务器将检查你提供的证书是否与user权限表中的SSL选项值相匹配。如果不匹配,这次连接将被拒绝。

在通过这几项验证之后,MySQL服务器将建立起这次连接并进入访问控制机制的第二阶段。如果建立起来的是一条加密连接,以后的通信都将被加密。

在第二阶段,你每发出一条查询命令,MySQL服务器就要去检查一遍各有关权限表,看你是否有足够的权限来执行这个查询。(如果user权限表里还有资源管理数据列,MySQL服务器还将检查你是否已经超出了`max_questions`和`max_updates`限制;MySQL会在检查你的操作权限之前去做这件事——如果你已经超出了那些限制,MySQL就不必浪费时间去检查你的操作权限了。)访问控制机制的第二阶段将一直持续到断开与MySQL服务器的连接为止。

下面,我们将对MySQL服务器用来匹配权限表记录项与外来客户连接请求和数据库查询命令的有关规则做进一步的介绍。这包括权限表中的权限作用范围数据列值的类型、来自不同权限表的权限值将如何组合以及某给定权限表里各有关字段的检索次序。



### 1. 权限作用范围数据列的内容

不同的权限作用范围数据列对哪些类型的值是合法的、MySQL将如何对这些值做出解释等方面都有自己的一套规则。有几个权限作用范围数据列必须是文本值，不允许使用通配符或其他特殊符号，但大多数权限作用范围数据列都允许使用通配符或其他特殊值。

- **Host** Host数据列的值可以是一个主机名或一个IP地址数字。特殊值localhost代表本地主机，它将匹配你在连接MySQL服务器时给出的主机值localhost、127.0.0.1或者（当你在一个基于Windows NT的系统使用命名管道来连接MySQL服务器时）“.”主机值。不过，localhost不能匹配MySQL服务器主机的真实名字或IP地址。比如说，假设本地主机的名字是cobra.snake.net，它的user权限表里有两个都对应于用户bob但Host值分别是localhost和cobra.snake.net的记录项。那么，在UNIX或Windows系统上，Host值是localhost的记录项将在用户bob使用以下命令来连接MySQL服务器时得到匹配：

```
% mysql -p -u bob -h localhost
% mysql -p -u bob -h 127.0.0.1
```

此外，在Windows系统上，Host值是localhost的记录项还将在用户bob使用以下命令来连接MySQL服务器时得到匹配：

```
C:\> mysql -p -u bob -h .
```

localhost连接在UNIX系统上使用UNIX套接字，在Windows系统上使用TCP/IP。127.0.0.1连接在两个平台上都使用TCP/IP。“.”连接在Windows系统上使用命名管道。

Host值是cobra.snake.net的记录项将在用户bob在本地主机cobra.snake.net上使用该主机的名字（即cobra.snake.net）或该主机的IP地址来连接MySQL服务器时得到匹配。在这两种情况下建立的连接都使用TCP/IP。

MySQL允许在Host值里使用SQL模式字符“%”和“\_”作为通配符，其含义与用法与它们在查询命令中的LIKE操作符下相同。（注意：配合REGEXP操作符使用的正则表达式不允许用做Host值。）这两个SQL模式字符既可以用在主机名里，也可以用在IP地址数字里。比如说，%.kitebird.com能匹配kitebird.com域中的任何主机，%.edu能匹配教育机构类网站中的任何主机。类似地，192.168.%能匹配B类子网192.168中的任何主机，192.168.3.%能匹配C类子网192.168.3中的任何主机。

如果Host值是单个的“%”字符，它将匹配任何主机，与之对应的那位用户就能从任何地方连接MySQL服务器。权限表里的空白Host值与单个“%”的含义相同——但db权限表是个例外，db权限表里的空白Host值表示“请检查host权限表里的信息”；具体情况见下一小节。）

从MySQL 3.23版本开始，还可以在权限表里给出网络编号的同时用一个掩码来指定客户IP地址的哪些位必须与你在权限表里给出的网络编号匹配。比如说，192.168.128.0/255.255.255.0指定了一个24位的网络编号，这个编号能匹配IP地址的前24位的值等于192.168.128的任何一台客户主机。

- **User** 用户名必须是文本值或空白。如果是空白，则能匹配任何名字，代表“匿名用户”；如果不是空白，则只能与客户给出的用户名精确匹配。比如说，把单个的“%”字符用做user值并不意味着空白——它将精确地与用户名“%”相匹配，这一点千万要注意。



在MySQL服务器利用user权限表对来自客户端的连接请求进行验证的时候，如果第一个得到匹配的记录项有一个空白的user值，MySQL服务器就会认为该客户是一个匿名用户。

- Password Password值是空白或非空白，而且不允许使用通配符。空白的Password值并不表示将匹配任何口令，相反，它表示用户必须不给出口令。

权限表里的Password值被存储为经过加密的密文而非普通的文本。如果把一个明文口令存放到Password数据列里，用户将无法连接！GRANT语句和mysqladmin password命令会自动地对口令进行加密，如果你打算使用INSERT、REPLACE、UPDATE或SET PASSWORD语句去直接修改权限表，就一定要使用PASSWORD('new\_password')来写出口令值，千万不要只给出'new\_password'。

- Db 在db和host权限表里，Db值既可以是文本值，也可以使用SQL模式字符“%”或“\_”作为通配符。“%”或空白值将匹配任意的数据库名。columns\_priv和tables\_priv权限表里的Db值只能是数据库名字的文本值并将精确匹配客户给出的名字，不允许是匹配模式和空白值。
- Table\_name和Column\_name 这两个数据列里的值只能是数据表或数据列名字的文本值并将精确匹配客户给出的名字，不允许是匹配模式和空白值。

某些权限作用范围数据列是区分字母大小写情况的，另一些则不是，如表12-4所示。需要特别注意的是，Db和Table\_name值总是区分字母大小写的，而查询命令中的数据库名和数据表名则往往要根据MySQL服务器在其上运行的文件系统来决定是否需要区分字母大小写情况。（一般说来，UNIX系统上的MySQL服务器要区分数据库名和数据表名中的字母大小写情况，Windows系统上的MySQL服务器则不区分。）

表12-4 权限表中的权限作用范围数据列的字母大小写敏感性

数据列	字母大小写敏感性
Host	否
User	是
Password	是
Db	是
Table_name	是
Column_name	否

#### user权限表里的口令是如何存储的

MySQL服务器在把口令保存到user权限表之前会先用PASSWORD()函数对它进行加密，这就有效地杜绝了以明文形式保存的口令会泄露给在user权限表上拥有读权限的用户的可能性。有很多人认为PASSWORD()函数与UNIX操作系统是使用同一种算法对口令进行加密的，但事实并非如此——MySQL与UNIX使用的是不同的加密算法，其相似之处只有它们都是单向和不可逆的。这意味着：即使你以UNIX口令作为MySQL口令，加密后得到的密文也不一定匹配。如果你在自行编写应用程序时想进行UNIX加密，就应该使用CRYPT()函数而不是PASSWORD()函数。（如果你想知道在自行编写应用程序时都有哪些加密方法可用，请参阅本书附录C。）

## 2. 数据库查询命令的权限是如何验证的

每发出一条查询命令，MySQL服务器都要检查你是否已经超出资源限额，如果没有，它将继续检查你是否有足够的权限来执行这个查询命令。资源限额由user权限表中的max\_questions和max\_updates值确定，但只有版本足够新的MySQL权限表才有这两个数据列。MySQL服务器将依次查看user、db、tables\_priv和columns\_priv权限表以验证你的操作权限，直到它能确定你具备或者不具备有关权限为止。具体过程如下：

1) 在接收到查询命令时，MySQL服务器将首先检查与这次连接相匹配的user权限表记录项，看你都有哪些全局级权限。如果你的全局级权限足以让你执行这个查询，MySQL服务器将执行之。

2) 如果你的全局级权限不够，MySQL服务器将继续检查db权限表中的匹配记录项。它会把 db权限表里找到的权限添加到你的全局级权限中。如果如此“积累”出的权限足以让你完成这次查询，MySQL服务器将执行之。

3) 如果把你的全局级和数据库级权限组合起来之后仍不足以让你执行这个查询命令，MySQL服务器将按先tables\_priv权限表、再columns\_priv权限表的顺序继续进行检查。

4) 如果在把所有的权限表都检查过一遍之后仍没能替你“积累”出足够的权限，MySQL服务器就将拒绝执行你的这个查询命令。

我们可以把MySQL服务器对权限表中的权限组合过程表示为如下所示的布尔不等式：

```
user OR db OR tables_priv OR columns_priv
```

有些读者可能会问，不是有5个权限表吗？为什么这里只用到了4个？这可问到点子上了，MySQL服务器其实是按下面这个表达式去检查操作权限的：

```
user OR (db AND host) OR tables_priv OR columns_priv
```

这里之所以先给出了一个比较简单的表达式，是因为大多数MySQL安装根本不会用到那个更复杂的表达式，而这又是因为出现在较复杂的表达式中的host权限表根本不受GRANT和REVOKE语句的影响，host权限表只有在你使用INSERT、UPDATE等语句直接对它进行操作时才会发生变化。也就是说，如果你在用户账户的管理方面采用的是以GRANT和REVOKE语句来进行的“正常”策略，你的host权限将根本不受影响，所以完全可以忽略它。

不过，如果你真的想使用host权限表，它的工作原理如下：

1) 当MySQL服务器检查数据库级权限时，它先去查看客户在db权限表里的记录项。如果Host数据列的值是空白，它的意思是“请到host权限表去查出哪些主机可以访问这个数据库”。

2) 在查看host权限表的时候，MySQL服务器会根据db权限表记录项中的Db数据列值去进行相关检索。如果host权限表登记项与客户主机不匹配，则不授予数据库级权限。如果某个记录项的Host数据列值与客户发起这次连接时所在的主机相匹配，MySQL将组合db权限表记录项和host权限表记录项以获得该客户的数据库级权限。

注意，db权限表和host权限表里的数据库级权限是按逻辑与（and）方式组合的，这意味着只有某项权限同时出现在db权限表记录项和host权限表记录项里时，客户才能具备这项权限。这

样，可以在db权限表记录项中授予一组基本权限，然后使用host权限表记录项为特定主机有选择地禁用一些权限。比如说，可以先允许网络中的所有主机都能访问某个数据库，但关闭那些不太安全的主机的数据库级权限。

以上内容可能会让大家觉得权限检查是个很复杂的过程，尤其是在考虑到MySQL服务器将为客户发出的每一个查询命令都进行这种权限检查的时候。其实，这个过程是相当快的，因为MySQL服务器并不是从各有关权限表里去读取权限控制信息，它会在启动时把权限表的内容读到内存里，然后利用内存里的信息去检查各个查询命令的权限。这大大提高了权限检查工作的性能，但同时也产生了一个必须引起高度重视的副作用：你对权限表内容的直接修改将不会被MySQL服务器注意到。

比如说，如果你是通过用INSERT语句往user权限表里插入一条新记录的办法来创建一个新MySQL用户的，新创建出来的用户将无法连接MySQL服务器。这类问题往往会让初出茅庐的MySQL管理员困惑不解（有些老手也不见得明白！），但解决的办法却很简单：在权限表的内容被你直接修改之后，让MySQL服务器把它们重新加载一次就行了。可以发出FLUSH PRIVILEGES语句、执行mysqladmin flush-privileges或mysqladmin reload来完成这项工作。

如果你是用GRANT、REVOKE或SET PASSWORD语句去设置或修改用户账户的，就不必让MySQL服务器重新加载权限表。MySQL服务器会把这些语句转换为对权限表的修改操作并在完成有关操作后自动地把权限表的内容重新加载到内存里去。

### 3. 权限作用范围数据列的匹配次序

MySQL服务器会按某种特殊的顺序对权限表中的记录项进行排序，然后通过依次遍历各记录项的办法去匹配来自客户的连接或查询请求并用第一个匹配到的记录项去验证客户的权限。因此，如果你想成为一名合格的MySQL管理员，就必须知道MySQL是如何对权限表（尤其是user权限表）进行排序的。这是做好MySQL安全工作的关键之一。

当MySQL服务器读取user权限表中的内容时，它会按Host和User数据列里的值对记录项进行排序。先按Host数据列进行排序，所以Host值相同的记录项将被排序操作集中到一起再按User值进行排序。不过，排序操作并不是严格地按字母表顺序进行的，字母表顺序只是部分地起作用。MySQL使用的排序原则是：文本值优先于模式值，较为具体的模式值优先于较为模糊的模式值。比如说，假设你正在从主机boa.snake.net去连接某个MySQL服务器，如果这个MySQL服务器上的user权限表有两个Host值，分别是boa.snake.net和%.snake.net的记录项，你将与第一个记录项（即Host值是boa.snake.net的那个记录项）匹配。类似地，Host值是%.snake.net的记录项将优先于Host值是%.net的记录项、Host值是%.net的记录项又将优先于Host值是%的记录项。IP地址的匹配也是按这种方式进行的。对于一个从IP地址为192.168.3.14的主机去连接MySQL服务器的客户，以下Host值都能与之匹配，但其优先顺序却如下所示：

```
192.168.3.14
192.168.3.%
192.168.%
192.%
%
```

### 12.2.3 一个与权限有关的难题

了解MySQL服务器在它验证连接/查询请求时对user权限表里的记录项进行排序的顺序有着非常重要的意义，这一点在本小节将要介绍的情况里体现得尤为突出。至少根据它在MySQL邮件列表上出现的频率，可以相信这个示例中的问题及其解决办法适用于很多新安装的MySQL软件。情况是这样的：一位MySQL管理员新安装了一套MySQL数据库系统，于是在新系统的user权限表里就有了对应于root和匿名用户的默认记录项。这位管理员给root账户设置了口令，但没有给匿名用户设置口令。现在，这位管理员要为一台需要能够从几台不同的主机去连接这个MySQL服务器的用户创建一个新账户。在用来创建这个账户的GRANT语句里把主机部分设置为“%”无疑是最简单的办法，而这应该使这位用户能够从任何地方去连接MySQL服务器：

```
GRANT ALL ON sampdb.* TO 'fred'@'%' IDENTIFIED BY 'cocoa';
```

这条GRANT语句的本意是把sampdb数据库上的所有权限全都授予用户fred并允许他从任何一台主机去连接MySQL服务器，可实际效果却不尽如人意：虽然用户fred能够从其他任何主机去连接MySQL服务器，但在这台MySQL服务器主机上却无法连接！比如说，当用户fred从boa.snake.net主机连接时，连接成功了：

```
% mysql -p -u fred -h cobra.snake.net sampdb
Enter password: cocoa
mysql>
```

但当用户fred登录到MySQL服务器主机cobra.snake.net并试图连接MySQL服务器时，虽然他给出了正确的口令，连接却失败了：

```
% mysql -p -u fred sampdb
Enter password: cocoa
ERROR 1045: Access denied for user: 'fred@localhost' (Using password: YES)
```

如果user权限表里有MySQL安装过程替你默认创建的匿名用户记录项（这个记录项的用户名部分是空白），就会遇到这个问题。在UNIX系统上，user权限表里的默认记录项是由mysql\_install\_db脚本初始化出来的；适用于Windows系统的MySQL发行版本通常会带有一个已经预初始化好了的user权限表。那么，用户fred在MySQL服务器主机cobra.snake.net上的连接动作为什么会失败呢？因为当MySQL服务器在检查用户fred的权限时，user权限表里一个对应于匿名用户的记录项先于对应于用户fred的记录项被匹配上了——被匹配的匿名用户记录项要求用户在连接时不给出口令（而不是给出口令cocoa），这导致了口令不匹配。

为什么会这样呢？要想把这个问题弄明白，我们先要从MySQL权限表的初始化设置情况和MySQL服务器用user权限表记录项去验证客户连接的过程说起。当在主机cobra.snake.net上用mysql\_install\_db脚本对权限表进行了初始化之后，user权限表记录项中的Host和User值将如下所示<sup>①</sup>：

① user权限表详细的初始化过程见第11.1节。



Host	User
localhost	root
cobra.snake.net	root
localhost	
cobra.snake.net	

前两个记录项允许root用户以localhost或cobra.snake.net为主机名去连接本地主机上的MySQL服务器，后两个记录项则允许用户以匿名方式连接到本地主机上的MySQL服务器。在MySQL管理员用前面所示的GRANT语句创建出fred账户之后，user权限表将包含以下记录项：

Host	User
localhost	root
cobra.snake.net	root
localhost	
cobra.snake.net	
%	fred

但这些记录项在这里的排列顺序并不是MySQL服务器在验证连接请求时使用的顺序——在验证工作开始之前，MySQL会按“先Host值，再User值；先较为具体的值，再较为模糊的值”的原则对记录项进行排序，排序结果如下：

Host	User
localhost	root
localhost	
cobra.snake.net	root
cobra.snake.net	
%	fred

请看，Host值都是localhost的两个记录项排在了一起，root项因为比空白值更具体而排在了前面。类似地，Host值都是cobra.snake.net的两个记录项也排在了一起。同时，因为这4个记录项的Host值里都没有通配符，所以它们都排在了Host值里有一个%字符的fred项的前面。请注意，按照这种排序规则，对应于匿名用户的两个记录项都将排在fred项的前面。

于是，当用户fred试图连接本地主机上的MySQL服务器时，无论他把主机名写成localhost还是写成cobra.snake.net，都会有一个对应于匿名用户的记录项（User值是空白）先于fred的记录项（Host值里有%字符）得到匹配。因为匿名用户的空白Password值与fred的口令cocoa无法匹配，连接就失败了。反过来说，如果用户fred不给出口令，他反而能从本地主机连接上MySQL服务器，但他将被验证为一名匿名用户而不会获得fred@%账户的各种权限。

由此可以得出这样一个结论：虽然用通配符来为某个用户创建一个能从多个主机连接



MySQL服务器的账户很方便，但这位用户却很可能在本地主机连接MySQL服务器时因为对应于user权限表里匿名用户的记录项而遇到麻烦。

那么，如何解决这个问题呢？有两个办法。其一，再为用户fred设置一个账户，把新账户的主机值明确地设置为localhost，如下所示：

```
GRANT ALL ON sampdb.* TO 'fred'@'localhost' IDENTIFIED BY 'cocoa';
```

如此设置之后，user权限表中的记录项将被排序为如下所示的样子：

Host	User
localhost	fred
localhost	root
localhost	
cobra.snake.net	root
cobra.snake.net	
%	fred

现在，当用户fred从本地主机连接时，与之对应的localhost项将在对应于匿名用户的记录项之前得到匹配；而当他从另外一台主机连接时，与之对应的%项将得到匹配。为用户fred创建两个记录项的缺点是：如果你今后想改变他的权限或口令，就必须同时修改这两个记录项才行。

第二个办法是把user权限表里对应于匿名用户的记录项全部删掉，但REVOKE语句不能实现这一目的——它只能收回权限，不能从user权限表里删除账户记录项，所以只能使用DELETE命令，如下所示：

```
% mysql -u root mysql
mysql> DELETE FROM user WHERE User = '';
mysql> FLUSH PRIVILEGES;
```

如此设置之后，user权限表中的记录项将被排序为如下所示的样子：

Host	User
localhost	root
cobra.snake.net	root
%	fred

现在，当用户fred从本地主机连接时，因为user权限表里已经不再有会在他前面被匹配到的记录项，他就能连接成功了。

一般说来，如果MySQL管理员想让自己的日子好过一些，最好把权限表里的匿名用户项全都删掉——有的观点认为，这些记录项的用处不大，麻烦却不少。

这一小节讨论的问题虽然比较特殊，但却有着较为普遍的意义：如果某个账户的权限在使用效果上与你预期的不一样，请检查各有关权限表，看其中有没有在因为Host值更为具体而会在该账户之前得到匹配的记录项。如果有的话，就很可能是问题的原因——或者需要把该用户的记录项设置得更为具体，或者需要为该用户再增加一个用来处理特殊情况的记录项。

### 12.2.4 应该避免的权限表风险

本小节将向大家介绍一些在进行授权时应该注意的事项。

首先，避免创建匿名的用户账户。匿名账户本身通常不具备引起直接破坏的权限，但只要别人能够利用它们进入你的数据库系统，就肯定能收集到不少有价值的信息，比如你的数据库系统里都有哪些数据库和哪些数据表等。

第二，找出没有口令的账户，删除它们或者给它们设置上口令。为了找出这类账户，要在mysql数据库中使用下面这个查询：

```
mysql> SELECT Host, User FROM user WHERE Password = '';
```

第三，如无必要，在创建账户时就不要在Host值里使用模式字符。允许某给定用户用来连接MySQL服务器的主机越多，别人冒充这位用户来攻击数据库系统的攻击点也就越多。

第四，对于超级用户权限，能不授予，就不授予。最好不要用user权限表进行授权。在user权限表里激活的权限都是全局级的，它们或者会使有关用户有权影响MySQL服务器的运行情况，或者会使他们有权访问任何一个数据库里的任何一个数据表。应该利用其他的权限表把用户的权限限制在给定的数据库、数据表或数据列上。

第五，不要把mysql数据库上的权限——尤其是允许进行各种写操作的权限——授予给别人，因为它包含着权限表。如果有权访问mysql数据库的用户还能够对权限表进行修改，他就能非法获得其他任何数据库上的访问权限。从效果上讲，如果你让某个用户有权修改mysql数据库里的数据表，就等于是把全局级的GRANT OPTION权限授予给了他——要知道，能直接修改权限表就等于能发出任何GRANT语句。

第六，对待GRANT OPTION权限要特别谨慎。拥有不同权限但都拥有GRANT OPTION权限的两位用户能够通过相互授权而使双方都获得更多的权限。

第七，FILE权限非常具有“杀伤力”，不要轻易把它授予别人。下面这个例子能让你明白一位具备FILE权限的用户都能做哪些事：

```
CREATE TABLE etc_passwd (pwd_entry TEXT);
LOAD DATA INFILE '/etc/passwd' INTO TABLE etc_passwd;
```

执行完这两条语句后，这位用户只需再发出下面这条SELECT语句就能看到MySQL服务器主机上的UNIX口令文件/etc/passwd里的内容了：

```
SELECT * FROM etc_passwd;
```

LOAD DATA语句中的文件名/etc/password可以被替换为MySQL服务器主机里的任何一个对全体用户可读的文件。如果还允许这位具备FILE权限的用户从一个远程主机来连接MySQL服务器，这位用户就能通过网络访问MySQL服务器主机上的文件系统的很多文件。

如果MySQL数据目录上的访问模式设置得不严密，就会留下让具备FILE权限的用户进入别人数据库的安全漏洞。这也是建议大家把MySQL数据目录的内容设置成只能由MySQL服务器读取的原因之一。要知道，如果对应于某个数据表的文件对所有用户都是可读的，那么，不仅在本地主机上有登录账户的用户能够读取它们，任何一位通过网络连接到MySQL服务器且具备FILE权限的客户端用户也能读取它们！具体过程如下：

1) 创建一个数据表，让这个数据表有一个LONGBLOB类型的数据列：

```
USE test;
CREATE TABLE tmp (b LONGBLOB);
```

2) 利用这个数据表依次读入你想“偷”的各个文件的内容。比如说，假设另一个用户在数据库other\_db中有一个名为x的MyISAM数据表，这个数据表将被表示为x.frm、x.MYD和x.MYI等三个文件。下面这些命令将把这三个文件的内容读入并拷贝到test数据库里去：

```
LOAD DATA INFILE './other_db/x.frm' INTO TABLE tmp
  FIELDS ESCAPED BY '' LINES TERMINATED BY '';
SELECT * FROM tmp INTO OUTFILE 'x.frm'
  FIELDS ESCAPED BY '' LINES TERMINATED BY '';
DELETE FROM tmp;
LOAD DATA INFILE './other_db/x.MYD' INTO TABLE tmp
  FIELDS ESCAPED BY '' LINES TERMINATED BY '';
SELECT * FROM tmp INTO OUTFILE 'x.MYD'
  FIELDS ESCAPED BY '' LINES TERMINATED BY '';
DELETE FROM tmp;
LOAD DATA INFILE './other_db/x.MYI' INTO TABLE tmp
  FIELDS ESCAPED BY '' LINES TERMINATED BY '';
SELECT * FROM tmp INTO OUTFILE 'x.MYI'
  FIELDS ESCAPED BY '' LINES TERMINATED BY '';
```

3) 执行完上面这些语句之后，test数据库目录里也将包含有三个名为x.frm、x.MYD和x.MYI的文件。换句话说，test数据库现在包含有你从other\_db数据库“偷”来的数据表x的一份副本。

为防止别人也像这样偷取用户们的数据表，请按第12.1.2节的有关步骤为MySQL数据库目录的内容设置好访问模式。如果想再保险点，还可以在启动MySQL服务器时用--skip-show-database选项禁止MySQL用户使用SHOW DATABASES和SHOW TABLES语句。这样，他们就不能看到他们无权访问的数据库和数据表的清单了。

如果把MySQL服务器运行在root用户名下，FILE权限的危险性就更大了。把MySQL服务器运行在root用户名下原本就不妥当，再加上FILE权限，风险就更大了：root用户有权在文件系统中的任何地方创建新文件，这使得具备FILE权限的用户——哪怕他们是从某个远程主机连接到MySQL服务器的——也能这样做。虽说MySQL不允许你去创建一个已经存在的文件，但即便是创建一个原本不存在的文件也有可能改变MySQL服务器主机的运行情况或者形成安全漏洞。比如说，假设MySQL服务器运行在root用户名下，如果服务器主机中的/etc/resolv.conf、/etc/hosts.equiv、/etc/hosts.lpd、/etc/sudoers等文件中的某一个或某几个不存在，具备FILE权限的用户就能利用MySQL服务器把它们创建出来，而这可能会让服务器主机的行为发生巨变。为避免这类问题，千万不要把mysqld程序运行在root用户名下。（另请参见第11.2.1节。）

第八，只把PROCESS权限授予可信任的MySQL账户。具备PROCESS权限的用户能够用SHOW PROCESSLIST语句去查看MySQL服务器正在执行的查询命令的文本——如果利用这一点去嗅探其他用户，偷看到他人的私密信息只是迟早的事。

第九，不要把RELOAD权限授予无关的用户。具备RELOAD权限的用户将有权发出各种FLUSH和RESET语句，一旦使用不当，就会造成以下问题：

- 如果你把MySQL服务器的变更日志或二进制变更日志配置成一系列以数字编号的文件，

那么，每一条FLUSH LOGS语句都会去创建一个新的变更日志或二进制变更日志文件。如果某位具备RELOAD权限的用户在短时间内多次执行了这条语句，就会导致MySQL服务器创建出大量的文件来。

- 具备RELOAD权限的用户将有权执行FLUSH PRIVILEGES或FLUSH USER-RESOURCES语句，这些语句将重新加载权限表并使资源管理计数器重新从零开始计数——这将使资源配额管理机制形同虚设。
- FLUSE TABLE语句将使MySQL服务器把它的数据库缓存区内容写到磁盘上，频繁地发出这条语句将使MySQL服务器因为无法利用这个缓存区而损害其性能。类似地，频繁发出RESET QUERY CACHE语句将使MySQL服务器因为无法利用其查询缓存区而损害其性能。
- RESET MASTER LOGS将使镜像机制中的主服务器删除它所有的二进制变更日志，而不管它们是否仍在被使用，这将使你无法保证镜像机制的数据完整性。

第十，ALTER权限的某些用法可能出乎你的预料。比如说，假设你规定用户user1只能访问数据表table1，不允许他访问table2。可如果user1（或其他用户）具备ALTER权限，就能轻易地绕过你做出的规定——只要用ALTER TABLE语句把table2改名为table1就行了。

### 12.2.5 不用GRANT语句创建MySQL账户

如果你的MySQL是3.22.11或更早的版本，就不能用GRANT（或REVOKE）语句去管理各个MySQL账户和它们的访问权限，但可以利用INSERT语句等去直接修改权限表的内容。如果你知道GRANT语句是如何修改权限表的，用INSERT语句去实现各种GRANT语句的效果就是一件很容易的事情了。

GRANT语句对权限表的修改过程如下：

- 当你发出一条GRANT语句时，MySQL服务器将在user权限表里创建一个记录项并把你给出的用户名、主机名和可能会有的口令分别记录在User、Host和Password数据列里。此外，如果你还在GRANT语句里设定了全局级权限，MySQL还将把那些权限记录到user权限表记录项的各有关权限数据列里。
- 如果你在GRANT语句里设定了数据库级权限，你给出的用户名和主机名将被记录到db权限表记录项的User和Host数据列里去。你在GRANT语句中给出的数据库名将被记录到Db数据列里，而你在GRANT语句中给出的权限将被记录在各有关的权限数据列里。
- 数据表级和数据列级的权限的记录办法与此大同小异：MySQL将在tables\_priv和columns\_priv权限表里创建记录项，并把用户名、主机名、数据库名以及相应的数据表名和数据列名记录到相应的数据列里去，你在GRANT语句中给出的权限将被记录到各有关权限数据列里。

根据以上描述，我们就能在不使用GRANT语句的前提下完成GRANT语句所能完成的任何事。但要记住，在直接修改权限表之后，一定要让MySQL服务器重新加载权限表才能让你做出的修改生效。可以用mysqladmin reload命令强制MySQL重新加载权限表<sup>①</sup>。如果忘了重新加载

① mysqladmin flush-privileges命令和FLUSH PRIVILEGES语句也能让MySQL服务器重新加载权限表。可是，如果你不使用GRANT语句的原因是你的MySQL服务器版本太旧而不支持这条语句的话，那它也就同样不支持mysqladmin flush-privileges命令或FLUSH PRIVILEGES语句。



权限表，你就会惊讶于MySQL服务器为什么会不按你的想法去做事。

下面这条GRANT语句使用了ON \*.\*子句来设定全局级权限。它将创建出一个超级用户账户，这个账户具备所有的权限——把权限转授给其他用户的能力也包括在内：

```
GRANT ALL ON *.* TO 'ethel'@'localhost' IDENTIFIED BY 'coffee'
WITH GRANT OPTION;
```

这条语句将在user权限表里创建出一个ethel@localhost记录项，并会把user权限表里的所有权限数据列都设置为“激活”——user权限表是存放超级用户（全局级）权限的地方。下面是能够完成同样事情的INSERT语句：

```
INSERT INTO user VALUES('localhost','ethel',PASSWORD('coffee'),
    'Y','Y','Y','Y','Y','Y','Y','Y','Y','Y','Y','Y');
```

这条INSERT语句非常难看！根据你具体使用的MySQL版本，你可能会发现它根本不起作用——这条语句需要user权限表有14个权限数据列才能执行成功，这个数字是GRANT语句最早被实现时user权限表的权限数据列的个数。但因为权限表的结构会时不时地发生一些变化，所以不同MySQL版本下的user权限表在权限数据列个数方面可能会有一些出入。因此，应该先用SHOW COLUMNS语句把user权限表都包含有哪些权限数据列的情况弄清楚，再对INSERT语句做出相应的调整。还要注意的，GRANT语句能自动对口令进行加密，INSERT语句却不会；必须明确地使用PASSWORD()函数对你在INSERT语句里给出的口令进行加密。

下面这条GRANT语句也创建了一个超级用户账户，但只授予了它一项权限：

```
GRANT RELOAD ON *.* TO 'flush'@'localhost' IDENTIFIED BY 'flushpass';
```

还记得这条语句吗？我们在第11章见过它。这个账户的用途是对日志文件进行维护，其权限仅限于对日志文件进行刷新。与这条GRANT语句等价的INSERT语句要比上面那条INSERT语句简单点，因为它只涉及到一个权限数据，其他的权限数据将被设置为它们的默认值（'N'）：

```
INSERT INTO user (Host,User,Password,Reload_priv)
VALUES('localhost','flush',PASSWORD('flushpass'),'Y');
```

数据库级权限要用ON db\_name.\*子句来设置，不能使用ON \*.\*子句：

```
GRANT ALL ON sampdb.* TO 'boris'@'localhost' IDENTIFIED BY 'ruby';
```

因为这些权限不是全局级的，所以它们不会被存放到user权限表里。不过，为了模拟出这条GRANT语句的效果，有必要在user权限表里创建一个记录项以便用户boris能够连接MySQL服务器。也就是说，数据库级权限需要你同时创建一个user权限表记录项和一个db权限表记录项才能被正确地设置出来：

```
INSERT INTO user (Host,User,Password)
VALUES('localhost','boris',PASSWORD('ruby'));
INSERT INTO db VALUES('localhost','sampdb','boris',
    'Y','Y','Y','Y','Y','Y','N','Y','Y','Y');
```

第二条INSERT语句中的'N'值对应于GRANT OPTION权限——如果把这个数据列设置为'Y'，



这条INSERT语句的执行效果将相当于一条带有WITH GRANT OPTION子句的数据库级GRANT语句。

要想设置数据表级或数据列级权限，就要用INSERT语句对tables\_priv或columns\_priv权限表进行操作。不过，如果你的MySQL版本没有GRANT语句，也就没有这两个权限表，因为它们都始见于同一个MySQL版本。如果你的MySQL有这两个权限表（因而也就有GRANT语句可用），但就是想以手动方式去对它们进行操作，请记住这样一点：这两个权限表里的权限数据列都只有一个，不要再使用好几个权限数据列去设置账户的各种权限了——需要把想要激活的权限构造成一个SET类型的值，再把这个SET值插入到tables\_priv.Table\_priv或columns\_priv.Column\_priv数据列里去。比如说，如果想激活某个数据表上的SELECT和INSERT权限，就应该把相应的tables\_priv记录项中的Table\_priv数据列设置成'Select, Insert'。

如果想改变某个现有MySQL账户的权限，要用UPDATE而不是INSERT语句去添加或者收回有关的权限。

如果想彻底删除某个账户，就要用DELETE语句把各个权限表里对应于该账户的记录项全部删除才行。比如说，如果想彻底删除mike@%.snake.net账户，就要发出以下语句：

```
DELETE FROM user WHERE User = 'mike' AND Host = '%.snake.net';
DELETE FROM db WHERE User = 'mike' AND Host = '%.snake.net';
DELETE FROM host WHERE User = 'mike' AND Host = '%.snake.net';
```

如果你凑巧还有tables\_priv或columns\_priv权限表，就要再多发出两条语句：

```
DELETE FROM tables_priv WHERE User = 'mike' AND Host = '%.snake.net';
DELETE FROM columns_priv WHERE User = 'mike' AND Host = '%.snake.net';
```

如果不想通过发出查询命令的办法去直接修改权限表，可以使用MySQL发行版本自带的mysqlaccess和mysql\_setpermissions脚本。

## 12.3 建立加密连接

UNIX系统上的MySQL 4提供了对SSL加密连接的支持。在默认的情况下，具备SSL支持机制的MySQL软件允许客户对是否要使用加密连接进行选择。（一般说来，普通的非加密连接有着较高的性能；加密连接虽然安全但速度较慢，因为对数据进行加密会加重系统的计算负担。）此外，MySQL数据库系统的管理员还可以用GRANT语句规定某些账户必须使用加密连接。

需要提醒大家注意的是，在用UNIX套接字文件、命名管道或IP地址127.0.0.1（本地主机的网络回馈接口）建立的对本地主机的连接上使用SSL没有什么意义。SSL的真正价值体现在所传输的信息会经过某个你怀疑正被人嗅探的网络时。

要想在MySQL服务器与客户程序之间建立起加密的SSL连接，必须满足以下几个条件：

- user权限表里必须有与SSL有关的数据列。
- 你的MySQL服务器和客户程序都已经编译有OpenSSL支持机制。
- 在启动MySQL服务器时用有关选项告诉它到哪儿能找到其证书文件和密钥文件；这些文件是建立加密连接所不可缺少的。

- 要想让某个客户程序建立加密连接，必须在调用这个客户程序时用有关选项告诉它到哪儿能找到其证书文件和密钥文件。

下面，我们将对以上过程做进一步的讨论。

首先，要想使用SSL加密连接，mysql数据库里的user权限表必须有第12.2.1节里提到的SSL数据列——如果你安装的是MySQL 4.0.0或更高的版本，user权限表就应该已经包含有这些数据列了；如果你是从老版本升级到MySQL 4.x版本的，这些数据列很可能尚不存在，需要运行mysql\_fix\_privilege\_tables脚本来升级权限表。

其次，你的MySQL发行版本必须已经把OpenSSL编译在其中的了。你既可以设法弄一份编译有OpenSSL的二进制发行版本，也可以自行编译MySQL软件的源代码。对于后一种情况，必须先安装好OpenSSL（可以从www.openssl.org站点下载到必要的软件包），再在运行MySQL软件的编译配置脚本configure时明确地给出--with-vio和--with-openssl选项并完成编译工作。在启动了具备OpenSSL支持机制的MySQL服务器之后，可以通过用mysql连接它并发出下面这个查询的办法来检查这个MySQL服务器是否支持SSL：

```
mysql> SHOW VARIABLES LIKE 'have_openssl';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| have_openssl  | YES   |
+-----+-----+
```

如果你没看到YES，就说明SSL支持没有被正确激活。

在正确地激活了MySQL的SSL支持机制之后，MySQL服务器和它的客户（程序）就可以利用加密连接来通信了。为建立加密连接，服务器端和客户端都要准备好三个文件，即：

- 证书签发机构（Certificate Authority，简称CA）证书。所谓“证书签发机构”指的是一个值得信赖的第三方，它的证书将被用来验证客户端和服务端提供的证书。CA证书可以从某个商业机构购买，也可以自行生成。
- 证书文件，连接的两端向另一端证明自己身份的文件。
- 密钥文件，用来对在加密连接上传输的数据进行加密和解密。

MySQL服务器端的证书文件和密钥文件必须首先安装。在sampdb发行版本的ssl目录里有几个供大家试用的样本文件：

- ca-cert.pem——CA证书。
- server-cert.pem——MySQL服务器的证书。
- server-key.pem——MySQL服务器的公共密钥。

先把这几个文件拷贝到MySQL服务器的数据目录里去，然后在它启动时会去读取的某个选项文件（比如UNIX系统上的/etc/my.cnf文件）中的[mysqld]选项组里加上几条选项，这些选项的作用是给出证书文件和密钥文件的路径名。比如说，如果数据目录是/usr/local/mysql/data，有关选项将如下所示：

```
[mysqld]
ssl-ca=/usr/local/mysql/data/ca-cert.pem
ssl-cert=/usr/local/mysql/data/server-cert.pem
ssl-key=/usr/local/mysql/data/server-key.pem
```

也可以把证书文件和密钥文件放在其他地方,但应该确保那个目录只能由MySQL服务器访问。修改完选项文件之后,重新启动MySQL服务器。

现在,客户程序仍能使用未加密的连接去连接MySQL服务器。如果你想让某个客户程序建立和使用加密连接,就要把供这个客户程序使用的证书文件和密钥文件也准备好。smpdb发行版本的ssl目录也提供了几个这样的文件:可以使用同一份CA证书文件(ca-cert.pem),客户端的证书文件和密钥文件的文件名分别是client-cert.pem和client-key.pem。先把这些文件拷贝到你个人账户下的某个目录里,然后在登录目录下的.my.cnf文件里加上几条选项以便让客户程序知道它们都放在哪里。比如说,假想建mysql建立和使用加密连接,就得先把这些SSL文件拷贝到登录目录/u/paul里,再把以下各行添加到.my.cnf文件里去:

```
[mysql]
ssl-ca=/u/paul/ca-cert.pem
ssl-cert=/u/paul/client-cert.pem
ssl-key=/u/paul/client-key.pem
```

可以如上所示去设置你自己的账户。注意,应该确保你的证书文件和密钥文件只能由你本人去访问。修改完.my.cnf文件之后,调用mysql程序并发出一个\s或STATUS命令,如果你看到的是类似于下面这样的输出(有SSL行),就说明连接是加密的:

```
mysql> \s
-----
./mysql Ver 12.10 Distrib 4.1.0-alpha, for apple-darwin5.5 (powerpc)
Connection id:          1
Current database:
Current user:           root@localhost
SSL:                    Cipher in use is EDH-RSA-DES-CBC3-SHA
...
```

还可以用如下所示的数据库查询命令来查看与SSL有关的MySQL服务器状态变量的设置情况:

```
SHOW STATUS LIKE 'Ssl%';
```

把与SSL有关的选项写在[mysql]选项组里将使mysql默认使用SSL连接,如果把这些选项改为注释或者从选项文件里删掉了它们,mysql将使用普通的非加密连接。此外,如果像下面这样调用mysql程序,它将忽略与SSL有关的选项:

```
% mysql --skip-ssl
```

如果想让其他客户程序也使用SSL加密连接,请把[mysql]选项组里的SSL选项拷贝到供它们专用的选项组里去。注意,如果把这些选项放在了通用的[client]选项组里,那些不能使用SSL连接的客户程序就都将执行失败——把SSL选项放在[client]选项组里不见得是个好主意。

如果不想在选项文件里列出SSL选项,也可以把它们写在命令行上。比如说,可以在登录目录里像下面这样调用mysql程序(注意,必须把这个命令行敲在同一行上):

```
% mysql --ssl-ca=ca-cert.pem --ssl-cert=client-cert.pem
--ssl-key=client-key.pem
```

不过，如果要经常使用SSL加密连接，这种在命令行上给出SSL选项的做法就太麻烦了。

以上讨论是在有关账户能够对是否使用SSL进行选择的基础上做出的。还可以规定某个账户不得使用未加密连接，即要求它必须使用SSL。对新建账户和现有账户均可做出这样的设置。

先说新建账户。像往常一样发出一条GRANT语句，但要增加一个REQUIRE子句来规定该账户必须遵守某些限制。比如说，假设你想创建一个名为laura的用户，他将从主机rat.snake.net连接到主机cobra.snake.net上的MySQL服务器以访问finance数据库。如果只要求他建立的连接是加密的就行，请使用下面这条语句：

```
GRANT ALL ON finance.* TO 'laura'@'rat.snake.net'
IDENTIFIED BY 'moneymoneymoney'
REQUIRE SSL;
```

如果还想更安全点，可以使用REQUIRE X509子句。这样，用户laura在连接时就必须提供一份合法有效的X509客户证书（这个证书文件对应于ssl-cert选项）才行，但只要这份证书是合法有效的，它的内容是什么无关紧要。如果你想做出更多的限制性规定，可以在REQUIRE子句里组合使用CIPHER、ISSUER或SUBJECT等限定词。CIPHER规定了加密连接必须使用哪一种加密密码。ISSUER和SUBJECT则分别规定了客户证书必须是签发自哪一个CA机构和证书的持有者必须是谁。这些限定词对已经是合法有效的证书做出了更细致的规定，即只有全部满足这些规定的证书才能用来建立SSL加密连接。请看下面这条GRANT语句，它要求客户证书必须由指定CA机构签发，而且必须使用EXP1024-RC4-SHA密码：

```
GRANT ALL ON finance.* TO 'laura'@'rat.snake.net'
IDENTIFIED BY 'moneymoneymoney'
REQUIRE ISSUER '/C=US/ST=WI/L=Madison/O=sampdb/OU=CA/CN=sampdb'
CIPHER 'EXP1024-RC4-SHA';
```

如果想修改某现有账户让它今后必须使用SSL连接，就要使用格式如下所示的GRANT USAGE语句，其中的require\_options代表想要设置的SSL属性：

```
GRANT USAGE ON *.* TO 'user_name'@'host_name' REQUIRE require_options;
```

GRANT USAGE ON \*.\*不会改变现有账户的现有权限，所以这类语句将只修改与SSL有关的属性。

可以用GRANT USAGE语句的REQUIRE NONE子句来撤销对某个现有账户必须使用SSL连接的规定：

```
GRANT USAGE ON *.* TO 'user_name'@'host_name' REQUIRE NONE;
```

sampdb发行版本中的证书文件和密钥文件完全能够让你建立起加密连接。不过，因为谁都能获得它们，所以用它们建立起来的加密连接在安全方面多少要打些折扣。在用这些文件检查完SSL能够正确工作之后，最好能把它们替换成你自己生成的有关文件。自行制作证书文件和密钥文件的具体步骤请参见sampdb发行版本中的ssl/README文件。当然，也可以考虑购买一份商业化证书。



## 第13章 MySQL数据库的备份、维护和修复

如果MySQL从你第一次安装好它以后就一直平稳地运行，那当然是最理想的了。但由于各种原因，从停电到硬件故障到MySQL服务器非正常关停（比如用kill -9终止MySQL服务器或出现机器崩溃等），而发生问题。这些你大都无法控制的事件往往会对数据库里的数据表造成破坏，典型情况是在数据表的写操作期间发生意外而导致它无法完成。在这一章里，将向大家介绍一些规避风险和万一发生灾难后的应对措施，涉及数据库备份技术、数据表检查和修复技术以及数据丢失后的恢复技术等多个方面。本章还将介绍一些用来把数据库转移到另一个服务器的数据库拷贝技术，这种事情经常会发生，并且与数据库备份技术有很多相似之处。

为预防可能发生的问题，应该采取以下措施：

- 激活MySQL服务器的自动恢复能力。
- 制定一份数据库备份计划。一旦发生最坏情况并需要去面对重大的系统灾难时，你将需要备份来进行恢复工作。还应该激活二进制日志，它们记载着你在备份之后又对数据库的内容进行过哪些修改。因激活二进制日志而导致的额外开销非常之小（约1%），所以没有理由不激活之。
- 有计划地安排一些预防性的维护工作，定期对数据表进行检查。数据表的日常检查工作能帮助你及时发现和纠正那些小问题而不是让它们变得更糟糕。

万一遇到数据表损坏或数据丢失问题，请按以下原则处理：

- 对数据表进行检查，对怀疑出现问题的进行修复。MySQL的数据表修复能力能应付很多小破坏。
- 如果对数据表进行的检查和修复仍不能使MySQL服务器恢复运行，就要用备份和二进制变更日志来进行数据恢复工作了。先用备份把数据表恢复到当初进行备份时的状态，再用二进制变更日志把数据表恢复到灾难发生之前的状态。

用来完成以上任务的工具包括MySQL服务器本身的能力和几个MySQL发行版本自带的工具程序：

- 在启动时，MySQL服务器能够自动检查数据表是否出了问题并能自动修复它发现的很多问题。当你在崩溃之后重新启动MySQL服务器时，这种能力将非常有用。有些检查是自动进行的，另一些检查可以用启动选项来激活。
- mysqldump和mysqlhotcopy程序能帮你制作数据库备份，这些备份将用来恢复数据库。
- 可以用CHECK TABLE和REPAIR TABLE等SQL语句让MySQL服务器去执行几种数据表维护和修复操作。mysqlcheck工具程序为这些SQL语句提供了一个命令行操作界面。
- myisamchk和isamchk工具程序也能对数据表进行检查并对它们进行多种修复。

有些程序，比如mysqlcheck和mysqldump，需要与MySQL服务器配合使用，它们是通过先连接上MySQL服务器再发出SQL语句的办法去告诉MySQL服务器要执行什么样的数据表检查或



备份操作的。另一些程序，比如myisamchk和isamchk，则直接在用来表示数据表的文件上进行操作。不过，因为MySQL服务器在运行时也要访问这些文件，这类工具程序其实是MySQL服务器的竞争者，所以必须采取措施防止它们相互干扰。比如说，当用myisamchk程序去修复某个数据表的时候，一定要保证MySQL服务器不会在此期间也去访问它，做不到这一点往往会导致比你正试图修复的问题更加严重的问题！

本章所涉及的几项管理任务，从制作备份到进行数据表修复，都需要与MySQL服务器合作，所以我们将如何与MySQL服务器进行协调开始讲起，然后讨论如何做好未雨绸缪的准备工作，最后介绍如何在必要时使用各种数据库恢复技术。

在UNIX系统上，当需要直接对MySQL数据目录中的数据表文件或其他文件进行操作时，应该登录为MySQL管理员再进行这些操作，这将保证你有足够的权限去使用这些文件。在本书里，我们给这个账户起的名字是mysqladm。当然，以root用户身份也可以访问这些文件，但在这种情况下，一定要保证有关文件在你开始操作之前和完成操作之后的访问模式和属主是相同的。

本章所涉及的各项语句和程序的完整选项清单见附录D和附录E。

#### isamchk和myisamchk的关系

本章对用来检查和修复MyISAM数据表的myisamchk工具程序做了详细的介绍。另一个相关的工具程序是isamchk，它在功能和用法上与myisamchk程序相似，只是要用在ISAM数据表上而已。本章直接涉及isamchk程序的讨论内容并不多，这是因为MyISAM数据表有着更优异的性能和功能，所以人们已经很少使用ISAM数据表了。不过，如果你在实际工作中确实需要用到ISAM数据表，本章对myisamchk程序的介绍也大都适用于isamchk程序。只要记住下面几点即可：

- 把所有与myisamchk程序有关的讨论内容中的.MYD和.MYI文件（MyISAM数据表的数据文件和索引文件）替换为.ISD和.ISM文件（ISAM数据表的数据文件和索引文件）。
- 如果你在检查和修复数据表时用错了程序，有关程序会发出一条警告，但不会对数据表造成任何破坏。比如说，下面是一条用isamchk程序去检查当前目录中的MyISAM数据表的命令：

```
% isamchk *.MYI
```

因为isamchk程序只能用在ISAM数据表而不是MyISAM数据表上，所以你将看到一些警告信息，但MyISAM数据表不会受到任何破坏。

- myisamchk程序有一些isamchk程序不支持的选项，具体情况见附录E。

### 13.1 与MySQL服务器进行协调

某些管理员操作需要连接MySQL服务器并告诉它去做什么事情。比如说，如果想对某个MyISAM数据表进行数据完整性检查或修复，一种做法是发出CHECK TABLE或REPAIR

TABLE语句让MySQL服务器去做这项工作，即让MySQL服务器去访问代表着这个数据表的.frm、.MYD和.MYI文件。一般说来，这种做法是最好的。让MySQL服务器去执行这类操作的好处是数据表上的读写动作将由它来协调，用不着你去操心。

检查或修复数据表的另一种做法是执行myisamchk工具程序。但myisamchk程序将直接访问有关的数据表文件，有关操作不再通过MySQL服务器进行，这就需要由你来协调数据表上的读写动作。在myisamchk程序对某个数据表进行处理的同时，你必须阻止MySQL服务器去修改这个数据表。如果不这样做，数据表就可能因为myisamchk程序和MySQL服务器发生竞争而遭到破坏甚至无法继续使用。很明显，让MySQL服务器和myisamchk程序同时对同一个数据表进行写操作是不好的，但即使它们是一个在读而另一个在写也是不好的——正在读取数据表文件的程序会因为另一个程序对数据表的修改而被弄糊涂。

这种问题也会发生在其他场合。比如说，有些备份技术需要制作数据表文件的拷贝，而要想保证备份文件的一致性，就必须防止MySQL服务器在备份工作进行过程中去修改有关数据表的内容。再比如说，有些恢复技术需要用备份出来的拷贝去更换被破坏的数据表，在这种情况下，必须阻止MySQL服务器去访问正在被恢复的数据表。

不让MySQL服务器打扰你的办法之一是将它彻底关停——既然没有运行，它当然就不会去访问你正在处理的数据表。但MySQL管理员通常都不太愿意把自己负责的MySQL服务器彻底关停，因为这将使其他的数据库和数据表都不可用。本节所描述的步骤将帮助你对正在运行着的MySQL服务器和你正在使用的外部程序做好协调，让它们不会相互干扰。

要想与MySQL服务器进行协调，就必然要用到某种锁定机制。MySQL服务器本身带有两种锁定机制：

- 内部锁定机制，用来防止来自不同客户程序的查询请求相互混杂和干扰——比如防止来自某个客户程序的SELECT查询不会被来自另一个客户程序的UPDATE查询打断。
- 外部锁定机制，用来防止别的程序在它正在使用某些数据表文件的同时去修改这些文件。MySQL服务器的外部锁定机制是以你的操作系统在文件系统级的锁定能力为基础的。人们给MySQL服务器加上外部锁定机制的理由就是为了让它能够与诸如myisamchk之类的工具程序在数据表检查操作期间进行协调。然而，MySQL服务器的外部锁定机制在某些系统上工作得不太可靠——既然不能依赖外部锁定机制，就只能用它的内部锁定机制来谋求协调了。此外，MySQL服务器的外部锁定机制只能用来与那些不需要往数据表文件里写入数据的操作进行协调；如果某个操作既要读数据表文件又要写数据表文件，MySQL服务器的外部锁定机制就帮不上忙了。（比如说，如果你正在修复某个数据表而不是在检查它，就应该使用MySQL服务器的内部锁定机制。）

本节所介绍的方法仅适用于那些把不同的数据表分别表示为一组相关文件的数据表类型（比如MyISAM、BDB和ISAM数据表），它不适用于InnoDB数据表，因为所有的InnoDB数据表都被保存在构成InnoDB表空间的同一组文件里。

### 13.1.1 使用内部锁定机制防止两个操作相互干扰

本小节将向大家介绍如何利用MySQL服务器的内部锁定机制去阻止它在你某个数据表进

行操作时去访问这个数据表，其基本思路是先用mysql客户程序连接到MySQL服务器并发出一条LOCK TABLE语句去锁定你打算对之进行操作的数据表；然后，把mysql程序闲置在那里（即让它待在那儿什么也不做，而你刚锁定的那个数据表将一直处于被锁定的状态），这样你就可以对数据表文件进行任何操作了；等完成有关操作之后，再回到mysql程序去解除对数据表的锁定，让MySQL服务器知道那个数据表又能用了。

以哪种方式去锁定数据表要取决于你对数据表文件是进行读操作还是进行读/写操作。如果只是检查或者拷贝那些文件，以只读方式锁定它们就足够了。如果需要修改这些文件（比如需要对数据表进行修复或者需要用完好的文件替换被损坏的文件），就得以读/写方式去锁定它们。

需要用LOCK TABLE或UNLOCK TABLE语句去锁定数据表或解除其锁定。还要用FLUSH TABLE语句通知MySQL服务器把内存里的信息写入磁盘，在发出FLUSH TABLE语句之后，MySQL服务器需要重新打开数据表才能对之进行访问。我们在示例中使用的是FLUSH TABLE语句，它能根据你给出的数据表名字只把指定数据表在内存里的信息写入磁盘。如果你的MySQL版本低于3.23.23，就只能使用FLUSH TABLES语句，它不支持以数据表名字作为参数的做法，会把整个数据表缓存区全部写入磁盘。

必须在同一个mysql会话任务里执行所有的LOCK、FLUSH和UNLOCK语句。如果你在锁定某个数据表后退出了mysql程序，所设置的锁定就将被解除，而MySQL服务器将认为它又能使用那个数据表了；这意味着你对数据表文件的操作将不再安全。

同时打开两个窗口（一个用来运行mysql客户程序，另一个用来对数据表文件进行处理）来进行操作是最方便的。这使你能够在保持mysql程序运行的同时去做你想做的事情。如果没有窗口环境可用，在直接对数据表文件进行操作时，就只能利用shell的作业调度命令去挂起和恢复mysql程序的运行了。

#### 1. 以只读方式锁定某个数据表

下面将要介绍的步骤适用于只需对数据表文件进行读操作（如制作这些文件的拷贝或者检查其数据完整性）的场合。此时，以只读方式锁定有关的数据表就已经足够了；MySQL服务器将替你阻止其他客户程序去修改它，但仍允许其他客户程序从中读取数据。在需要对数据表做出修改的场合，不要使用这个办法。

1) 在窗口A里执行mysql程序，然后用以下语句申请一个读操作锁并把数据表在内存里的信息写入磁盘：

```
% mysql db_name
mysql> LOCK TABLE tbl_name READ;
mysql> FLUSH TABLE tbl_name;
```

读操作锁将阻止其他客户在你检查给定数据表的同时向其写入数据和修改它。FLUSH语句将使MySQL服务器关闭有关的数据表文件，而文件关闭操作将把缓存在内存里的有关信息写入磁盘。

2) 把mysql程序闲置在那儿，切换到窗口B去对数据表文件进行想进行的操作。比如说，可以用下面这条命令去检查一个MyISAM数据表：

```
% myisamchk tbl_name
```

3) 完成对数据表的处理之后, 切换回窗口A里的mysql会话任务并解除对数据表的锁定:

```
mysql> UNLOCK TABLE;
```

在对数据表进行处理的过程中, 可能会发现还需要对它做进一步的处理。比如说, 在用myisamchk程序检查某个数据表的过程中, 可能会发现一些需要纠正的问题。如果纠正工作需要你以读/写方式去锁定那个数据表, 就需要用下一小节介绍的步骤去锁定之。

## 2. 以读/写方式锁定某个数据表

下面将要介绍的步骤适用于需要对数据表文件的内容进行修改(比如修复某个数据表)的场合。此时, 必须申请一个写操作锁才能完全阻止MySQL服务器在你对数据表文件进行处理的时候去访问它。

为修复数据表而进行的锁定与为检查数据表而进行的锁定在步骤上差不多, 但有两个不同之处。第一, 必须以读/写方式进行锁定而不是以只读方式进行锁定。你将对数据表进行修改, 所以绝对不能让MySQL服务器去访问它。第二, 在处理完数据表之后必须再发出一条FLUSH TABLE语句。有些操作(比如用myisamchk程序去修复数据表)会创建一个新的索引文件, 如果你不发出FLUSH TABLE语句, MySQL服务器就不会知道它的存在。以读/写方式锁定某个数据表的具体步骤如下:

1) 在窗口A里执行mysql程序, 然后用以下语句申请一个写操作锁并把数据表在内存里的信息写入磁盘:

```
% mysql db_name
mysql> LOCK TABLE tbl_name WRITE;
mysql> FLUSH TABLE tbl_name;
```

2) 把mysql程序闲置在那儿, 切换到窗口B去对数据表文件进行想进行的操作。比如说, 可以用下面这条命令去修复一个MyISAM数据表:

```
% myisamchk --recover tbl_name
```

这个例子仅仅是个演示。你将发出的命令要取决于想进行的工作。(为预防万一, 最好先制作一份数据表文件的拷贝。)

3) 完成对数据表的处理之后, 切换回窗口A里的mysql会话任务, 再次发出FLUSH TABLE语句, 然后解除对数据表的锁定:

```
mysql> FLUSH TABLE tbl_name;
mysql> UNLOCK TABLE;
```

## 3. 以只读方式锁定所有数据表

要想阻止客户程序对任何一个数据表做出修改, 最简单的办法莫过于以只读方式锁定所有数据库里的所有数据表。可以用下面这条语句来做到这一点:

```
mysql> FLUSH TABLES WITH READ LOCK;
```

下面这条语句将解除这种锁定:

```
mysql> UNLOCK TABLES;
```

当你以只读方式锁定所有数据库里的所有数据表时, 其他客户只能从中读取数据, 但不能



进行修改。当你制作整个MySQL数据目录的拷贝时，用这个办法让MySQL服务器保持“沉默”是最合适的了。但从另一方面讲，这对想要修改数据表的其他客户来说不太友好，所以一定要在完成任务后立刻解除这种锁定。

### 13.1.2 使用外部锁定机制防止两个操作相互干扰

在某些场合，在直接对数据表文件进行操作的时候可以利用MySQL服务器的外部锁定机制去与MySQL服务器进行协调。比如说，如果你的系统支持MySQL服务器的外部锁定机制，myisamchk和isamchk程序就能利用这一机制去与MySQL服务器配合工作。但这仅限于只想对数据表文件进行读操作（比如对数据表进行检查）的时候，需要对数据表文件进行读/写访问（比如对数据表进行修复）的操作不能依赖于MySQL服务器的外部锁定机制。MySQL服务器的外部锁定机制是以操作系统的文件锁定机制为基础的，但因为myisamchk和isamchk程序在修复数据表时会先把数据表文件拷贝为一些新文件、等修复工作完成之后再新文件替换掉原来的数据表文件——MySQL服务器对新文件将一无所知，所以即便想利用文件锁定机制去协调它们对数据表的访问，你的努力也将徒劳无功。

从MySQL 4开始，在各种系统上，MySQL服务器的外部锁定机制都是默认禁用的。如果你确定这一机制在你的系统上能够正确工作，可以把它激活。但一般来讲，还是不要使用MySQL服务器的外部锁定机制为好——MySQL服务器的内部锁定机制更值得信赖。

如果你想知道你的MySQL服务器是否激活了外部锁定机制，可以去检查相应的MySQL服务器变量。这个变量的名字在MySQL 4及以后的版本中是skip\_external\_locking，在MySQL 4之前的版本里中是skip\_locking。可以用下面这个查询来查出它的值：

```
mysql> SHOW VARIABLES LIKE 'skip%locking';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| skip_external_locking | ON    |
+-----+-----+
```

根据skip\_external\_locking（或skip\_locking）变量的值，有以下两种选择：

- 如果skip\_external\_locking的值是ON，则表明MySQL服务器的外部锁定机制被禁用，MySQL服务器将无法知道myisamchk或isamchk程序是不是在访问数据表。此时，如果你需要在直接对数据表文件进行处理时让MySQL服务器保持运行，就必须通过它的内部锁定机制告诉它不要去访问你正在处理的数据表。根据打算进行的操作，请以只读方式或读/写方式锁定有关的数据表。
- 如果skip\_external\_locking的值是OFF，则表明MySQL服务器的外部锁定机制已被激活，这时就可以用myisamchk或isamchk程序对数据表进行只读性的操作（比如对数据表进行检查）。MySQL服务器和你使用的工具程序将协调它们对数据表的访问。但要注意两件事：首先，在执行myisamchk或isamchk程序之前，应该先用mysqladmin flush-tables命令把数据表缓存区里的内容写入磁盘；其次，必须保证不会有人在你对数据表文件进行处理的过



程中去修改相应的数据表。还要注意一点：如果你想修复某个数据表，因为你可能对它做出修改，所以就不应该使用MySQL服务器的外部锁定机制——应该使用MySQL服务器的内部锁定机制以读/写方式去锁定有关的数据表。

## 13.2 在灾难发生前做好准备工作

MySQL管理员的职责之一是保证他们的数据库完好无损。本节将向大家介绍一些这方面的基本策略：

- 充分利用MySQL服务器本身所具备的自动恢复能力。
- 定期对数据库进行备份。这样，万一数据库遭到了无可挽回的破坏，你也有最后的绝招。

定期对数据库进行预防性的维护也是一项很重要的工作，与此有关的技术将在第13.3节里介绍——因为它们要用到其中介绍的数据表维护工具。

### 13.2.1 充分利用MySQL服务器的自动恢复能力

MySQL服务器会在启动时对数据库进行自动恢复的能力是数据库完整性维护工作的第一道防线。在启动过程中，MySQL服务器会进行一些数据表检查工作以帮助人们解决早些时候因MySQL服务器或机器崩溃而导致的问题。MySQL服务器的启动过程能自动查出和纠正很多问题，在很多时候，只需像往常那样重新启动MySQL服务器，就能让它完成必要的纠错和修复工作。MySQL服务器可能采取的动作包括：

- InnoDB数据表处理程序（如果被激活了的话）能自动查出和纠正很多问题。比如说，对于记载在重做日志（redo log）里的事务（即那些已被提交但没来得及写入数据表的事务），它会重做（redo）一遍；对于记载在撤销日志（undo log）里的事务（即在崩溃发生时正在执行的未提交事务），它会进行回滚。这样做的结果是让InnoDB数据表总保持在完好无损的状态，各有关用户在崩溃发生前提交的所有事务都将毫不走样地反映在他们的InnoDB数据表的内容里。
- BDB数据表处理程序（如果被激活了的话）也能根据其日志文件的内容去进行相应的自动恢复工作。
- MyISAM处理程序也能自动进行检查和修复操作，其具体行为由MySQL服务器的--myisam-recover=level选项控制。选项值level或者为空，意思是“不进行检查”；或者是以下几个值中的一个或者多个（它们彼此以逗号分隔）：DEFAULT（与没有使用这个选项的效果相同）、BACKUP（如果自动恢复工作需要修改某个数据表，先为它创建一个备份）、FORCE（强行恢复，哪怕会因此而丢失多个数据行）和QUICK（快速恢复）。--myisam-recover选项最早出现于MySQL 3.23.25版本。

要是InnoDB或BDB数据表上的自动恢复工作因为一个无法恢复的问题而失败了，MySQL服务器就会在错误日志里写出一条消息后退出执行。如果你打算强行启动MySQL服务器以尝试某种手动恢复办法，请参阅第13.3.2节的“恢复InnoDB表空间或BDB数据表”小节。

MySQL服务器不会在启动时对ISAM数据表做任何检查，未来也不太可能会增加这一功

能——自从各方面都更出色的MyISAM数据表出现之后，MySQL在ISAM支持方面就再也没有什么大动作了。强烈建议你吧ISAM数据表转换为MyISAM数据表。可以用下面这样的ALTER TABLE语句把数据表转换为MyISAM格式：

```
ALTER TABLE tbl_name TYPE = MYISAM;
```

还可以使用mysql\_convert\_table\_format工具程序，它只需一条命令就能把某个数据库里的所有数据表全都转换为MyISAM格式。这个脚本程序是用Perl语言编写的，它要求你安装有DBI模块。如果你想知道这个脚本的具体用法，请用--help选项执行之。

如果你不想对ISAM数据表进行格式转换，可以设法在启动MySQL服务器之前用isamchk程序去检查它们。另外，如果你的MySQL服务器的版本早于3.23.25（早于--myisam-recover选项被引入之前），可以在启动MySQL服务器之前用myisamchk程序去检查MyISAM数据表。第13.3.1节的“定期进行预防性维护”小节对如何在MySQL服务器启动之前使用这些工具程序去进行数据表检查工作的问题进行了讨论。

### 13.2.2 备份和拷贝数据库

为预防数据表的丢失或损坏，对数据库进行备份非常重要。在发生严重的系统崩溃时，你肯定希望自己能够以最少的数据损失为代价把数据表恢复到崩溃发生之前的状态。类似地，如果某个用户不理智地发出了一条DROP DATABASE或DROP TABLE语句，他肯定会找上门来让MySQL管理员帮他进行数据恢复。有时候，甚至是MySQL管理员也会（比如试图用vi或emacs等编辑器去直接编辑某个数据表文件时）造成破坏。这对数据表来说肯定不是好事！

数据库备份技术在需要把一个数据库复制到另一个MySQL服务器去的时候也很有用。把数据库转移到运行在另一台主机上的MySQL服务器下是最常见的，但也可以把数据转移到运行在同一台主机上的另一个MySQL服务器去。当你安装了一个新版本的MySQL服务器并想用一些实际数据对它进行测试的时候，就很有可能会这样做。

备份的另一个用途是建立镜像服务器。建立从服务器的前几个步骤之一就是在某特定时刻给主服务器“拍一张照片”。这张“照片”就是对主服务器的备份，只要把它加载到从服务器里，就能让从服务器里的数据库与主服务器一模一样。此后，用户在主服务器上所做出的修改将通过标准的镜像协议被原样复制至从服务器上。建立镜像机制的具体步骤请参阅第11章。

对数据库进行备份主要有两种办法：一是使用mysqldump程序；二是使用mysqlhotcopy、cp、tar或cpio等程序来直接拷贝数据库文件。两种方法各有其优缺点，主要表现在：

- mysqldump程序与MySQL服务器要相互配合。直接拷贝法则是在MySQL服务器外部进行的文件拷贝操作，必须设法保证MySQL服务器在你拷贝数据表文件时不会去修改有关的数据表。当你通过对文件系统进行备份的办法来备份数据库时，也要注意同样的问题：在对文件系统进行备份的时候，如果某个数据表正在被修改，进入备份的数据表文件就可能与留在MySQL服务器主机里的数据表在内容上有细微差异，以后再恢复这个数据表也就没有意义了。不过，作为MySQL管理员，你在文件系统的备份方面可能没有什么发言权（这事归系统管理员负责），但在直接拷贝数据表的时候，你可是有权采取各种必要的措施

来保证MySQL服务器不会去访问有关数据表的。

- `mysqldump`程序比直接拷贝法要慢，因为转储操作需要通过网络来传输信息。直接拷贝法是在文件系统级进行的，信息不需要通过网络传输。
- `mysqldump`程序生成的是文本文件，很容易移植到其他的机器上，甚至可以移植到不同硬件结构的机器上，所以这种文件特别适合用来拷贝数据库。直接拷贝法生成的文件就不一定能移植到其他机器上了——这要取决于数据表的存储格式是否与具体的机器无关。ISAM数据表不能满足这一条件。比如说，把数据表文件从SPARC平台上的Solaris操作系统直接拷贝到SPARC平台上的Solaris操作系统不会有问题，但把它们从SPARC平台上的Solaris操作系统拷贝到Intel平台上的Solaris操作系统或Mac OS X操作系统就无法使用了。MyISAM和InnoDB数据表通常与具体的机器无关，这两类数据表的数据表文件都可以直接拷贝到运行在另一台有着不同硬件结构的机器上的MySQL服务器去。第3章已经对各种数据表类型的特点（包括它们的可移植性）做了详细的讨论。

无论选用的是哪一种备份方法，要想让今后的数据库恢复工作达到最佳效果，就必须遵守以下几个原则：

- 定期对数据库进行备份。制定一个计划，并严格按照这个计划行事。
- 激活MySQL服务器的二进制变更日志（详见第11.4节）。当需要在系统崩溃后去恢复数据库时，二进制日志能帮上大忙。先用备份文件把数据库恢复到当初对它进行备份时的状态，再去运行二进制变更日志里记载的查询命令就能把数据库恢复到崩溃发生时的状态。
- 给备份文件起的名字既要有规律，又要有意义。诸如`backup1`、`backup2`之类的名字并没有多大的意义；等真的要用它们去进行恢复操作的时候，你得浪费些时间才能把其中都有哪些东西搞清楚。用数据库的名字和备份日期来构造备份文件的文件名是个很不错的主意——比如说：

```
% mysqldump sampdb > /archive/mysql/sampdb.2002-10-02
% mysqldump menagerie > /archive/mysql/menagerie.2002-10-02
```

- 定期对备份文件进行失效处理以防止它们填满你的硬盘。可以用第11章介绍的日志文件轮转技术来做这件事，备份文件的失效处理工作完全可以按照同样的原则来进行。
- 使用文件系统的备份来备份你的备份文件。要知道，如果系统崩溃不仅让你失去了MySQL数据目录，而且还波及到了你用来存放数据库备份文件的硬盘，你就只能求助于文件系统的备份了。把日志也备份下来。
- 不要把备份文件和数据库放在同一个文件系统上。首先，这能避免备份文件挤占MySQL数据目录的可用空间。其次，如果用来存放备份文件的文件系统位于另一个驱动器上，还能降低因硬盘故障而导致的损失程度——除非两个驱动器同时发生故障，否则MySQL数据目录和备份文件总会有一个能“幸存”下来。

#### 1. 用`mysqldump`程序来备份和拷贝数据库

在用`mysqldump`程序去生成数据库备份文件时，备份文件将默认地被写为SQL格式，它由一系列CREATE TABLE和INSERT语句组成，CREATE TABLE语句负责创建被转储的数据表，INSERT语句则包含有该数据表中各数据行的数据。等以后重建数据库的时候，只要把

mysqldump的输出文件用做mysql程序的输入,就能把它们重新加载到MySQL数据库系统里去。  
(千万不要用mysqlimport程序去读取mysqldump程序的SQL格式的输出!)

可以把某个数据库整个地转储为一个文本文件,如下所示:

```
% mysqldump sampdb > /archive/mysql/sampdb.2002-10-02
```

输出文件的开头部分如下所示:

```
-- MySQL dump 9.06
--
-- Host: localhost      Database: sampdb
-----
-- Server version      4.0.3-beta-log
--
-- Table structure for table 'absence'
--
CREATE TABLE absence (
  student_id int(10) unsigned NOT NULL default '0',
  date date NOT NULL default '0000-00-00',
  PRIMARY KEY (student_id,date)
) TYPE=MyISAM;
--
-- Dumping data for table 'absence'
--
INSERT INTO absence VALUES (3,'2002-09-03');
INSERT INTO absence VALUES (5,'2002-09-03');
INSERT INTO absence VALUES (10,'2002-09-06');
...
```

这个文件的后续内容是更多的CREATE TABLE和INSERT语句。

备份文件的尺寸通常都比较大,所以你肯定想让它们小一些。一种办法是使用--opt选项,它将对转储过程进行优化并生成一个尺寸较小的文件:

```
% mysqldump --opt sampdb > /archive/mysql/sampdb.2002-10-02
```

还可以对转储文件进行压缩。比如说,下面这条命令将一边进行转储一边进行压缩:

```
% mysqldump --opt sampdb | gzip > /archive/mysql/sampdb.2002-10-02.gz
```

如果你觉得尺寸过大的转储文件难以管理,还可以分别转储数据库里的各个数据表;在mysqldump命令行上的数据库名后面给出相应的数据表名即可,mysqldump程序将只转储给定数据库里的给定数据表,而不是给定数据库里的所有数据表。如此转储的备份文件尺寸会小很多,管理起来自然就容易多了。下面这个例子将把sampdb数据库里的几个数据表分别转储为不同的文件:

```
% mysqldump --opt sampdb student score event absence > gradebook.sql
% mysqldump --opt sampdb member president > hist-league.sql
```

如果生成备份文件的目的是为了用它们去定期更新另一个数据库的内容,--opt选项就会很有用。--opt选项将自动激活--add-drop-table选项,后者将使mysqldump程序在备份文件里的每条



CREATE TABLE语句之前加上一条DROP TABLE IF EXISTS语句，这两条语句中的数据表名字完全一样。这样，当你把备份文件加载到第二个数据库时，就不会发生“数据表已经存在”的错误了。如果你想对另一个MySQL服务器进行测试却又不想把它设置为镜像机制中的从服务器，就可以利用这个技巧定期地把在第一个MySQL服务器上转储的数据库备份拿到第二个MySQL服务器上把数据加载进去。

如果只是为了把数据库转移到另一个MySQL服务器去，可能连创建备份文件都用不着：先在目标主机上创建出相应的数据库，再把源主机上的数据库经网络转储到目标主机去，同时使用一个管道让mysql程序直接读取mysqldump程序的输出。比如说，如果想用这个办法把sampdb数据库从本地主机拷贝到boa.snake.net主机上的MySQL服务器去，可以像下面这样做：

```
% mysqladmin -h boa.snake.net create sampdb
% mysqldump --opt sampdb | mysql -h boa.snake.net sampdb
```

如果你在本地主机上的MySQL账户不允许你去访问boa.snake.net上的MySQL服务器，但你在boa.snake.net上有一个允许访问它本身的账户，就可以利用ssh在那台主机上远程调用MySQL命令，如下所示：

```
% ssh boa.snake.net mysqladmin create sampdb
% mysqldump --opt sampdb | ssh boa.snake.net mysql sampdb
```

等你以后想要更新boa.snake.net主机上的sampdb数据库时，重复执行上面这条mysqldump命令就行了。

mysqldump程序还有其他一些你可能需要用到的选项：

- --flush-logs和--lock-tables选项的组合能在数据库检查工作中帮上你的忙。--lock-tables用来锁定你打算转储的所有数据表，--flush-logs选项用来关闭再重新打开日志文件。如果你的变更日志或二进制变更日志的文件名是以数字编号的，新日志将从你发出mysqldump --flush-logs之后开始记载对数据做出了修改的查询命令。这就能把日志同步到你进行备份时的时间。（在备份期间锁定所有数据表的做法对客户不太友好，但如果客户会在此期间去修改数据表的话，还是把它们都锁定了更稳妥。）
- 如果打算用--flush-logs选项把日志同步到进行备份的时间，最好把整个数据库都转储出来。在恢复操作期间，按数据库提取日志内容的做法是很常见的。如果你转储的是各个数据表，让日志检查点与备份文件保持同步就比较困难。（mysqldump程序没有能按数据表提取日志内容的选项，你只能自己去提取它们。）
- 在默认情况下，mysqldump会先把一个数据表的内容全部读入内存再写出去，但这并没有必要；事实上，如果你的数据表尺寸非常大的话，这种做法往往会成为失败的原因。可以用--quick选项让mysqldump每检索出一个数据行就立即把它们写到备份文件里去。要想进一步优化转储过程，可以用--opt选项代替--quick选项。--opt选项将激活其他几个能加快数据转储速度的选项。此外，转储文件里的信息会被写成将在数据加载工作中得到更快处理的格式。

用--opt选项进行备份可能是最常见的做法，因为它使备份速度加快。但使用--opt选项是有代价的：--opt优化的是备份工作而不是客户对被备份的数据库的访问。--opt选项会把你正



在转储的数据表全都锁定上，不让任何人对其中的任何一个进行修改。这种做法对数据库用户们的影响很容易看到：只要在数据库往常最忙的时候开始制作备份，用不了多一会儿，肯定会有人打电话来问你出了什么事。

- `--delayed`选项的作用恰好与`--opt`选项相反。这个选项将使mysqldump写入INSERT DELAYED语句而不是INSERT语句。如果你正在把一个转储文件加载到另一个数据库并想把加载操作对该数据库里的其他查询操作的影响降到最低，就要用`--delayed`选项来实现这一想法。
- 人们通常只在mysqldump命令行上给出一个数据库名，在它后面再可选地给出几个数据表名。如果想同时转储多个数据库，就要使用`--databases`选项。mysqldump将把你在命令行上给出的名字全部解释为数据库名并依次把这些数据库里的数据表全都转储出来。如果想转储某给定MySQL服务器里的所有数据库，可以使用`--all-databases`选项——不用给出任何数据库名或数据表名参数。如果打算把转储文件加载到另一个MySQL服务器去，请慎重对待`--all-databases`选项——转储文件里将包括mysql数据库里的各个权限表，而你通常并不想去替换另一个MySQL服务器的权限表。
- 当需要把一个数据库拷贝到另一台机器上时，`--compress`选项很有用，因为它能减少在网络上传输的数据量：

```
% mysqldump --opt sampdb | mysql --compress -h boa.snake.net sampdb
```

请注意，`--compress`选项是在与远程主机进行通信的命令行（例子中的mysql）里给出的，而不是在与本地主机进行通信的命令行（例子中的mysqldump）里给出的。这里所说的“压缩”指的是经网络传输的数据量；它不会导致在目标数据库里创建出被压缩过的数据表来。

- 在默认情况下，mysqldump将同时转储数据表的结构（CREATE TABLE语句）和数据表的内容（INSERT语句）。如果只想转储其中之一，请使用`--no-create-info`或`--no-data`选项。

mysqldump程序还有很多其他的选项，请查阅附录E中的更多内容。

## 2. 以直接拷贝法来备份和拷贝数据库

要备份数据库或数据表、但又不使用mysqldump的另一种方法是直接拷贝数据表文件。典型做法是用cp、tar或cpio等工具程序来完成这项工作。在使用直接拷贝法制作备份的时候，一定要确保没有人在使用那个数据表。如果你这边在拷贝、MySQL服务器那边在修改，拷贝出来的数据表就没有实际价值。保证拷贝完整性的最好方法是关停MySQL服务器，拷贝有关文件，然后重新启动MySQL服务器。如果你不想关停MySQL服务器，请按第13.1节中所述的有关步骤以只读方式锁定有关数据表。这将阻止MySQL服务器在你拷贝数据表文件时修改相应的数据表。

假定MySQL服务器已关停或以只读方式锁定了你要拷贝的数据表，下面的例子演示了如何用直接拷贝法把sampdb数据库整个地备份到备份目录。假设MySQL数据目录是/usr/local/mysql/data，如下所示：

```
% cd /usr/local/mysql/data
% cp -r sampdb /archive/mysql
```

各数据表可以像下面这样进行备份：

```
% cd /usr/local/mysql/data/sampdb
% cp member.* /archive/mysql/sampdb
% cp score.* /archive/mysql/sampdb
...
```

完成备份工作后，如果你刚才关停了MySQL服务器，现在要重新启动它；如果你刚才让MySQL服务器保持运行并锁定了数据表，现在要解除之。

直接拷贝法也能用来把数据库从一台机器拷贝到另一台机器去。比如说，可以使用scp而不是cp程序。假设主机boa.snake.net的数据目录是/var/mysql/data，以下命令将把sampdb数据库目录拷贝到那台主机上：

```
% cd /usr/local/mysql/data
% scp -r sampdb boa.snake.net:/var/mysql/data
```

注意，在用直接拷贝法把数据库拷贝到另一个主机时，需要注意以下几个问题：

- 两台机器必须有同样的硬件结构，或者将拷贝的数据表全都是可移植的数据表类型。要不然，目标主机上的数据表就可能出现非常奇怪的内容。
- 在两台主机上都必须阻止MySQL服务器在你拷贝文件的同时去修改它们。最安全的做法是先关停MySQL服务器，再去拷贝数据表文件。
- 如前所述，直接拷贝法最适用于像MyISAM和ISAM这样会把不同的数据表分别表示为数据目录里的一组文件的数据表类型。要想用在InnoDB数据表上，就得再多注意几个问题；请参见稍后的“对InnoDB表空间或BDB数据表进行备份”小节。

### 3. 用mysqlhotcopy工具程序制作备份

从3.23.11版本开始，在MySQL发行版本里多了一个能帮你制作数据库备份的Perl DBI脚本mysqlhotcopy程序。这个名字中的“hot”指的是备份工作将在MySQL服务器保持运行时进行，你不必关停它。

mysqlhotcopy的优点主要有以下几点：

- 比mysqldump要快，因为它直接拷贝文件，而不是像mysqldump那样要通过MySQL服务器才能工作。（这意味着你必须在服务器主机上运行这个脚本；它不能用来对远程MySQL服务器进行备份。）
- 使用方便，它能替你完成必要的锁定工作，阻止MySQL服务器对你正拷贝的数据表进行修改；它用的是MySQL服务器的内部锁定机制（参见第13.1节）。
- 能刷新日志。这使备份文件和日志文件的检查点能保持同步并使以后用备份进行恢复容易一些。

mysqlhotcopy有好几种调用方法。假想拷贝sampdb数据库，下列命令将在MySQL数据目录里创建一个目录sampdb\_copy，并把sampdb数据库目录里的文件拷贝到该目录里：

```
% mysqlhotcopy sampdb
```

如果想把数据库拷贝到某给定目录下的名为sampdb的目录里，就要在数据库名的后面给出那个目录。比如说，下面这条命令将把sampdb数据库拷贝到/archive/2002-09-12/sampdb目录里

去:

```
% mysqlhotcopy sampdb /archive/2002-09-12
```

如果你想了解mysqlhotcopy脚本各命令的用法,请在mysqlhotcopy命令行上增加一个-n选项。这个选项将使mysqlhotcopy运行在“不执行”状态,即只显示有关的命令但不实际执行之。

#### 4. 用BACKUP TABLE语句制作备份

BACKUP TABLE语句最早出现于MySQL 3.23.25版本,它是MySQL服务器自带的一种数据表备份手段,但仅适用于MyISAM数据表。这个语句的用法如下所示,tbl1等是想备份的数据表,关键字“TO”后面必须是MySQL服务器主机上的一个目录,备份文件将被放到这个目录里:

```
BACKUP TABLE tbl1, tbl2, tbl3 TO '/archive/sampdb';
```

这个目录必须存在且对MySQL服务器可写,你必须拥有那些数据表的FILE权限和SELECT权限。BACKUP TABLE语句会先把各有关数据表在内存里的信息写入磁盘,再把各数据表的.frm(定义)和.MYD(数据)文件从数据目录拷贝到指定的目录去。它不拷贝.MYI(索引)文件,因为它能用另两个文件重建出来。

BACKUP TABLE语句依次锁定每一个数据表,但每次只锁定一个数据表。这意味着,当对多个数据表进行备份时,到这条语句执行结束的时候,备份文件有可能出现和该数据表的实际状态有所不同的情况。假定你是在备份tbl1和tbl2,BACKUP TABLE语句只在备份tbl1时才锁住tbl1,因此在备份tbl2时,tbl1有可能被修改。这意味着当BACKUP TABLE语句结束时,tbl1的内容会和备份文件的内容不相同。为了保证备份文件的内容与各有关数据表的内容全都保持匹配,就要在BACKUP TABLE语句执行期间以只读方式锁定这些数据表,等完成任务后再发出UNLOCK TABLES语句释放之。比如说:

```
LOCK TABLES tbl1 READ, tbl2 READ;  
BACKUP TABLE tbl1, tbl2 TO 'backup_dir_path';  
UNLOCK TABLES;
```

用BACKUP TABLE语句备份的数据表可以用RESTORE TABLE语句重新加载到MySQL服务器里,请参见第13.3节。

#### 5. 对InnoDB表空间或BDB数据表进行备份

类似于其他数据表,InnoDB和BDB数据表可以用mysqldump转储。也可以使用直接拷贝法,但要注意以下几个问题:

- InnoDB数据表不使用分开的文件来表示(除了每个数据表有一个.frm说明文件外)。它们都存放在由一个或者多个大文件构成的InnoDB表空间里。要想直接拷贝InnoDB表空间,就必须拷贝其所有的组成文件。为确保InnoDB处理程序已经把缓存区中的事务都提交了,应该先关停MySQL服务器再拷贝它们。为完整起见,还应该把对应于各InnoDB数据表的所有.frm文件、InnoDB日志文件和用来设定表空间配置情况的选项文件都拷贝下来。(为选项文件制作拷贝的目的是:即便它丢失了,也能重新对表空间进行初始化。)

另一种方法是使用InnoDB Hot Backup工具程序,可从innodb.com上得到它。这是允许你在MySQL服务器保持运行时制作InnoDB备份的商业化工具。

- 要直接拷贝BDB数据表，就要把该MySQL服务器所管理的所有BDB数据表都拷贝下来，还必须拷贝BDB日志文件。这应该在MySQL服务器关停时进行。BDB处理程序要求其日志在MySQL服务器启动时必须存在，这意味着如果有必要去恢复BDB数据表，就必须提供它们的日志。BDB日志文件默认创建在MySQL数据目录里，其名字的形式是log.nnnnnnnnnnn，有一个10位数字的后缀。

#### 6. 使用镜像机制帮助制作备份

制作备份是非常重要的，但这会与MySQL管理员的职责形成冲突。一方面，你想把MySQL服务器的能力扩大到用户群的每一个人，包括允许他们进行数据库修改。而另一方面，为了恢复的目的，备份是最有用的，但要保证你的备份文件和日志文件的检查能够同步进行。这些目标是相互矛盾的，因为同步备份和日志检查的最好方法是制作备份时刷新日志，并要关停MySQL服务器或同时锁定所有数据表保证不发生修改。（比如说，将--opt选项用于mysqldump工具程序。）然而，不允许修改会减少备份期间客户对数据表的访问。

如果建立了一个镜像从服务器，就能帮助你解决这个矛盾。不是在主服务器制作备份，而是使用从服务器。你不必关停主服务器，否则在备份期间客户不能使用它。用SLAVE STOP语句挂起在从服务器上的镜像机制，并用从服务器制作备份。（如果使用直接拷贝备份法，则发出FLUSH TABLES语句。）随后，用SLAVE START语句重新启用镜像，从服务器将捕获备份期间主服务器所做的任何修改。根据备份的方法，甚至不必挂起镜像机制。比如说，如果只备份单个数据库，可以使用具有相应选项的mysqlhotcopy或mysqldump同时锁住所有数据表。在这种情况下，从服务器仍在运行，只是不会在备份期间对已被锁定的数据表做任何修改。当备份完成和锁定释放后，从服务器重新开始自动地进行修改处理。

#### 7. 使用备份给数据库更名

MySQL没有重新命名数据库的命令，但是可以进行这项工作。用mysqldump转储数据库，用新名字创建新的空数据库，然后把转储的文件重新加载到新数据库里去。此后，可以删除老数据库。下面的例子演示了如何把db1更名为db2：

```
% mysqldump db1 > db1.sql
% mysqladmin create db2
% mysql db2 < db1.sql
% mysqladmin drop db1
```

一个更简单的数据库更名办法是：关停MySQL服务器，重新命名数据库目录，然后重新启动MySQL服务器。不过，只有在数据库中没有BDB或InnoDB数据表时才能使用这种方法。这个方法之所以不适用于BDB数据表，是因为到每个数据表的路径名要在其.db文件中编码。这种方法之所以不适用于InnoDB数据表，是因为各个数据表的数据库名都存放在InnoDB表空间，不受数据库目录更名操作的影响。

无论选用哪种方法改变数据库的名字，都要记住用户对数据库的访问权限要受mysql数据库中的各种权限表控制。如果权限表里有与你打算更名的数据库有关的记录项，必须相应地参照新名称来调整那些记录项。对于db1更名为db2的数据库，其使用的语句如下所示：



```
mysql> UPDATE db SET Db = 'db2' WHERE Db = 'db1';  
mysql> UPDATE tables_priv SET Db = 'db2' WHERE Db = 'db1';  
mysql> UPDATE columns_priv SET Db = 'db2' WHERE Db = 'db1';  
mysql> UPDATE host SET Db = 'db2' WHERE Db = 'db1';
```

对于user权限表，不需要这样的语句，因为它没有Db数据列。

### 13.3 数据表修复和数据恢复

会让数据库受损的原因有很多，受损的轻重程度也有很大的变化。如果幸运，可能只有一两个数据表受到了轻微的破坏。（比如说，机器因为停电而关停。）对于这种情况，MySQL服务器往往能在恢复正常运行时修复受损的数据表。如果不幸，就得更换整个数据目录（比如说，如果硬盘发生故障并破坏了你的数据目录）。其他场合也可能需要进行数据库恢复，比如说当用户错误地丢弃了某个数据库或数据表，或者删除了数据表的内容。这些不幸事件的原因无论是什么，你都必须执行某种恢复：

- 如果数据表受到了破坏但没有丢失，先用CHECK TABLE语句或mysqlcheck或myisamchk工具程序试试能不能修复它们。如果能修复好的话，就不必求助于备份文件了。
- 如果数据表已丢失或无法修复，就需要用备份来恢复它们。

本节首先叙述数据表的检查和修复方法，可以用它们来解决很多不严重的问题。这包括与MySQL服务器进行必要的协调、如何制定预防性防护计划等等。然后讨论数据表彻底失去或已无法修复时如何恢复它们。

#### 13.3.1 检查和修复数据表

如果你怀疑数据表受到了破坏，请按以下步骤进行检查和修复：

- 1) 检查数据表是否出了问题。如果没查出问题，工作就完成了。如果查出了问题，就必须修复之。
- 2) 为预防万一，在修复工作开始之前请先给数据表文件做一份拷贝。虽然可能性不大，可万一你在修复工作中犯了无可挽回的错误，还能用拷贝恢复出一份受损的数据表并去尝试另一种修复方法。
- 3) 对数据表进行修复。如果修复成功，工作就完成了。如果修复不成功，就得用数据库备份和变更日志去恢复之。

该方法的最后一步假设你已执行了数据库备份和启用了二进制变更日志。如果没有这样做，那么就存在着危险。阅读本章前面的讨论，其叙述了如何制作备份。同样要阅读第11章，找出如何启用日志。不要因为你疏忽了保存恢复数据表所必要的信息而处于失去数据表后就不可挽回的局面。

可以使用在数据表文件上直接操作的myisamchk工具程序来检查或修复数据表，或者可以告知MySQL服务器使用CHECK TABLE或REPAIR TABLE语句来检查或修复数据表。（或者使用mysqlcheck工具程序，该程序连接到MySQL服务器，并发出这些语句。）使用SQL语句或mysqlcheck的优点是MySQL服务器为你做这项工作。如果运行myisamchk，必须保证当你在数



据表上工作时MySQL服务器不使用该数据表文件。

正如本章前面所述，如果在让MySQL服务器做这项工作和运行外部工具程序之间进行选择的话，则应该让MySQL服务器做这项工作，这样你不必担心数据表访问协调的问题。你也可能会决定使用外部程序，比如说myisamchk，主要理由如下：

- 当MySQL服务器关停时你可能使用它。而CREAT TABLE 和REPAIR TABLE需要MySQL服务器运行。
- 你可以告知myisamchk使用较大的缓冲区，使检查和修复操作进行得较快。这对数据表非常大时是有用的。
- myisamchk可以用于较早的MySQL服务器。CHECK TABLE和REPAIR TABLE语句分别是在MySQL 3.23.13和3.23.14版本中引入的，而myisamchk可用于3.23.0版本。（当然，随着时间的推移，比3.23.13还早的MySQL服务器已经很少使用，这些差别就变得无关紧要了。）

**注意** 下面对myisamchk程序的介绍也适用于isamchk。参见本章开头的“isamchk和myisamchk的关系”中的内容。

#### 1. 用myisamchk程序来检查和修复数据表

在使用myisamchk检查或修复数据表前，你可能想关停MySQL服务器，使其在你正使用数据表文件时不去访问它们。如果你想保持MySQL服务器运行，请参阅第13.1节。其中介绍了MySQL服务器的锁定机制，防止MySQL服务器在你对数据表进行检查或修复过程中使用它们。

##### (1) 调用myisamchk

myisamchk不会对数据表的存放位置做出任何假设，你必须在运行它时明确地给出你打算使用的数据表文件的路径名。最方便的做法当然是先进入数据表所在目录，所以，在调用myisamchk之前，应该先进入有关的数据库目录，然后告知它你要检查或修复哪个数据表并给出相关的选项：

```
% myisamchk options tbl_name ...
```

tbl\_name参数可以是数据表名，也可以是数据表的索引文件名。这意味着可以根据索引文件的扩展名用一个文件名模式去选取当前目录里的所有相关文件：

```
% myisamchk options *.MYI
```

如果你不想对原始数据表文件进行操作，可以把它们拷贝到另一个目录中，然后用那个目录中的拷贝进行工作。

##### (2) 用myisamchk检查数据表

myisamchk提供的数据表检查方法，根据它们如何详细地检查数据表而变化。为了执行通常的数据表检查，可使用下列命令中的任何一个：

```
% myisamchk tbl_name
% myisamchk --check tbl_name
```

不带选项的myisamchk的默认操作是--check，所以这些命令是等效的。

默认的检查方法已经足以查出绝大多数问题。如果它报告说没有错误，但你仍怀疑有损坏（也许因为查询结果看起来有问题），可以用--extend-check选项做更细致的检查。这可能相当慢，

但这种方法非常彻底：对数据表的数据文件中的每个记录，对索引文件中每个索引的相关关键字都一一检查，以保证其指向正确的记录。（myisamchk也支持--medium-check选项，执行适中的检查，这种做法彻底性稍微差一些，但比高级检查要快。）

如果带有--extend-check选项的检查报告为没有错误，你可以相信数据表是正常的。如果你对数据表还有疑问，则原因必定是在另外的地方。重新检查所有可能出问题的查询，核对它们是否书写正确。如果你认为问题出在MySQL服务器上，请考虑填写一份漏洞报告或者升级到更高的版本。

如果myisamchk报告数据表有错误，就应该尽量修复它。

### （3）用myisamchk修复数据表

数据表修复是一件非常麻烦的事情，事实的确如此，其细节往往相当偶然和专业。不过，只要遵守一般的原则和步骤，就会大大增加你能修复数据表的机会。一般来讲，开始用最快的修复方法来查看并修正损坏。如果发现还不够，则要逐步升级到更细致（但较慢）的修复办法。直到损坏修复好，或者已经到了不能进一步修复的地步。（在实际工作中，绝大多数故障用不着进入更细致、更慢的高级检查模式就能得到修复。）如果数据表无法修复，就要用备份去恢复它。用备份文件和日志文件去进行数据表恢复的具体步骤见本章后面的内容。

下面是对数据表进行标准检查的具体做法：

1) 先试试--recover选项能不能修复数据表，还可以加上--quick选项，即根据索引文件的内容去进行恢复，数据表的修复工作不涉及数据文件：

```
% myisamchk --recover --quick tbl_name
```

2) 如果问题仍然存在，去掉--quick选项后再次执行这条命令，这将允许myisamchk修改数据文件：

```
% myisamchk --recover tbl_name
```

3) 如果仍不奏效，试试--safe-recover修复模式。这比普通的恢复模式要慢，但能修复几个--recover模式无能为力的问题：

```
% myisamchk --safe-recover tbl_name
```

在执行这些命令时，myisamchk可能会在显示“Can't create new temp file: file\_name”（无法创建新的临时文件）形式的错误信息后退出运行。如果发生这种情况，给命令行加上--force选项并再执行一遍；--force选项将强行删除此前失败的修复工作所遗留下来的临时文件。

如果使用标准修复步骤修复数据表失败，索引文件或许失去或损坏难以修复，还有可能数据表说明文件丢失（尽管不大可能）。在上述任一情况下，必须更换受影响的文件，然后再次试用标准修复步骤。

为了给数据表t再生成索引文件，使用下列步骤：

1) 移入含有已损坏的数据表的数据库目录。

2) 把数据表的数据文件t.MYD移至一个安全区域。

3) 调用mysql客户程序执行下列语句来重新建立一个新的空白数据表，该语句使用数据表说明文件t.frm，从头生成新的数据和索引文件：

```
mysql> TRUNCATE TABLE t;
```

在MySQL 4.0版本以前使用DELETE语句替代：

```
mysql> DELETE FROM t;
```

4) 退出mysql, 把原来的数据文件移回至替换刚建立的新的空数据文件的数据库目录中。数据文件和索引文件现在不同步, 但是索引文件现在有一个合法的内部结构, MySQL服务器可以根据数据文件和数据表说明文件的内容进行解释和重新建立。

5) 重新进行标准的数据表修复。

为了恢复数据表说明文件t.frm, 先从备份文件中恢复它, 然后重新进行标准的修复。如果由于某些原因没有备份, 但你知道必须发出CREATE TABLE语句来建立数据表, 则还能修复它:

1) 移入含有已损坏的数据表的数据库目录。

2) 把数据表的数据文件t.MYD移至一个安全区域。如果要使用索引文件t.MYI, 则同样要移它。

3) 调用mysql, 并发出CREATE TABLE语句建立数据表。

4) 退出mysql, 把原来的数据文件移回至替换刚建立的新的数据文件的数据库目录中。如果在第2步中移动了索引文件, 同样要把它移回至数据库目录中。

5) 重新进行标准的数据表修复。

(4) 使myisamchk更快运行

myisamchk要花费较长时间来运行, 如果你使用较大的数据表或使用一种更细致的检查或修复方法就更是如此。你可以告知myisamchk在运行时使用更多的内存来加速这个过程, myisamchk有几个操作参数可以设定。最重要的几个变量控制着要使用的缓冲区大小:

变 量	意 义
key_buffer_size	用于存放索引块的缓冲区大小
read_buffer_size	用于读操作的缓冲区大小
sort_buffer_size	用于排序的缓冲区大小
write_buffer_size	用于写操作的缓冲区大小

为了找出myisamchk默认使用的这些变量是什么数值, 可以用--help选项运行它。为了指定不同的数值, 在命令行上使用--set-variable variable=value或-O variable=value。这些变量名可以简写成key、read、sort和write。比如说, 如果你有许多内存, 可以告知myisamchk使用512MB排序缓冲区以及1MB读和写缓冲区, 调用格式如下所示:

```
% myisamchk -O sort=512M -O read=1M -O write=1M other-options tbl_name
```

sort\_buffer\_size仅用于--recover选项 (不用于--safe\_recover), 在这种情况下, key\_buffer\_size不能使用。

## 2. 使用MySQL服务器检查和修复数据表

CHECK TABLE 和REPAIR TABLE语句提供了一个至MySQL服务器的数据表检查和修复能力的SQL接口。它们适用于MyISAM数据表, 在MySQL 3.23.29及以后的版本中, CHECK TABLE语句也适用于InnoDB数据表。

对于每个语句, 提供一个或多个数据表名称的清单, 后跟选项, 指出使用哪种检查或修复

方式。比如说, 下列语句对三个数据表执行中等程度的检查, 而且假设它们没有正确关闭:

```
CHECK TABLE tbl1, tbl2, tbl3 FAST MEDIUM;
```

下列语句试图以快速修复方式修复同样的数据表:

```
REPAIR TABLE tbl1, tbl2, tbl3 QUICK;
```

CHECK TABLE语句允许通过下列选项指定执行哪一种检查:

- CHANGED 不检查数据表, 除非它们没有正确关闭, 或者自从最后检查后又发生了改变。
- EXTENDED 执行深入检查。这是能得到的最彻底的检查, 因此是最慢的。它试图核对数据行和索引文件的完全一致性。
- FAST 不检查数据表, 除非它们没有正确关闭。
- MEDIUM 执行中等程度的检查。如果未指定选项, 这是默认值。
- QUICK 执行快速检查, 只扫描索引行, 不检查数据行。

在某些情况下, CHECK TABLE语句有可能会修改数据表。比如说, 如果数据表标记为损坏或没有正确关闭, 但该检查没有发现问题, CHECK TABLE将把数据表标记为正常。其改变仅包括修改内部标志。

REPAIR TABLE语句允许通过下列选项指定修复方式:

- EXTENDED 试图重新建立索引进行修复。(这就好像使用带有--safe\_recover选项的myisamchk。)
- QUICK 只快速修复索引。
- USE\_FRM 试图用数据表的.frm文件进行修复。该修复是根据数据表的说明重新建立索引。它基本上自动操作前述的步骤, 使用.frm文件在索引文件失去或不可用的情况下从数据表的说明中重新建立索引, 当索引失去或损坏时这可能是有用的。该选项已引入MySQL 4.0.2版本中。

没有选项的REPAIR TABLE语句执行标准修复, 如同myisamchk --recover所做的一样。

myisamcheck工具程序提供了一个CHECK TABLE及REPAIR TABLE语句的命令行界面。该程序连接到MySQL服务器, 并根据指定的选项发出相应的语句。这可用于MySQL 3.23.38及以后的版本中。它可以检查MyISAM数据表(而InnoDB数据表是在MySQL 3.23.29及以后的版本中才可用它来检查)。

一般来讲, 调用具有数据库名的myisamcheck工具程序, 可选地后跟一个或多个数据表名称。只使用一个数据库名称, myisamcheck则检查数据库中所有的数据表:

```
% mysqlcheck sampdb
```

数据库名称后跟数据表名称是仅检查这些数据表:

```
% mysqlcheck sampdb president member
```

如果指定了--databases选项, 所有跟随的名字被解释为数据库名称, 而且myisamcheck工具程序要检查每个数据库中的所有数据表:

```
% mysqlcheck --databases sampdb test
```



如果指定了--all-databases, myisamcheck就会检查所有数据库中的所有数据表。在这种情况下, 不用提供数据库或数据表名称的参数:

```
% mysqlcheck --all-databases
```

默认情况下, myisamcheck使用标准检查方法来检查数据表, 但支持允许你执行更专用型操作的选项。下面展示了某些myisamcheck选项及其对应的CHECK TABLE选项:

mysqlcheck选项	CHECK TABLE 选项
--check-only-changed	CHANGED
--extended	EXTENDED
--fast	FAST
--medium-check	MEDIUM
--quick	QUICK

myisamcheck也可以执行数据表修复操作。下面列出了某些myisamcheck选项及其对应的REPAIR TABLE选项:

mysqlcheck选项	REPAIR TABLE 选项
--repair	无选项 (执行标准修复操作)
--repair --quick	QUICK
--repair --extended	EXTENDED

myisamcheck工具程序提供了一种便于直接发出CHECK TABLE和REPAIR TABLE语句的方法, 因为这些语句需要明确给出要检查或修复的每一个数据表的名称。myisamcheck对你给出相应数据表名称的语句查看其数据表名称和组成。

### 3. 定期进行预防性维护

除了启用自动恢复和建立备份步骤, 正如本章前面所述的那样, 应当考虑制定一个预防性维护计划。这有助于自动检测故障, 以便能采取措施纠正它。通过定期地检查数据表, 将会减少借助备份的可能性。从用于运行MySQL服务器的账户的crontab文件中调用cron作业, 这是最容易实现的。比如说, 如果你以mysqladm用户运行MySQL服务器, 就能从mysqladm用的crontab文件中设置定期检查。(参见第11章中有关设置cron作业的内容。)

为了自动定期地当MySQL服务器在线时检查MyISAM和InnoDB数据表, 可使用mysqlcheck工具程序。假定要从mysqladm用户所用的crontab文件中调用mysqlcheck, 则对那个文件增加一个记录项, 如下所示:

```
0 3 * * 0 /usr/local/mysql/bin/mysqlcheck --all-databases
--check-only-changed --silent
```

上面所示的命令分成两行, 但是应该完全写在一行上。该记录项告知cron在每个星期日早上3时运行mysqlcheck。该时间可以改变或按需要编排计划。

--all-databases选项使得mysqlcheck检查所有数据库内的所有数据表。这是一个很容易的方法并可以最大效率地使用它。--check-only-changed选项告诉mysqlcheck跳过在最后一次成功检查后没有修改过的数据表, 而--silent选项阻止输出, 除非数据表中有错误 (只要一个作业真正产生任何输出, cron作业一般产生一个邮件信息。对于未找到故障的数据表检查作业来讲, 很少有理由接收邮件)。但要注意, 如果数据库中有不知道如何检查的数据表, 即使用--silent也会



从mysqlcheck中得到某些诊断输出。

如果你有ISAM或MyISAM数据表，而且使用的是较早版本的MySQL，它不包括mysqlcheck，则可以使用isamchk和myisamchk来检查数据表。为了自动执行数据表检查，可以编写一个简单的脚本程序，针对给定目录下的所有MyISAM和ISAM数据表运行这些工具程序。下列脚本程序说明了进行这项工作的一种方法：

```
#!/bin/sh
# chk_mysql_tables.sh - check all MyISAM/ISAM tables under a given directory

# Argument 1: directory pathname

if [ $# -ne 1 ]; then
    echo "Usage: $0 dirname" 1>&2
    exit 1
fi

# Change location to directory, check tables under it.
# Notes:
# - Prior to MySQL 3.23.22, change --check-only-changed to --fast
# - isamchk does not support --check-only-changed or --fast

cd $1
if [ $? -ne 0 ]; then
    echo "Cannot cd to $1" 1>&2
    exit 1
fi
find . -name "*.MYI" -follow -print \
    | xargs myisamchk --silent --check-only-changed
find . -name "*.ISM" -follow -print \
    | xargs isamchk --silent
```

为了在这个脚本程序中只使用一个数据表检查工具程序，可省略掉不需要的命令。比如说，如果没有ISAM数据表，则可以省略isamchk命令，因为在这种情况下，find将不提供它的文件名，而isamchk将显示一个使用信息，指示你必须指定某些参数。

用chmod +x制作可执行的脚本程序，然后以数据目录的路径名调用它，检查所有的MyISAM和ISAM数据表：

```
% chk_mysql_tables.sh /usr/local/mysql/data
```

为了只检查给定数据库中的数据表，用相应的数据库目录的路径名调用脚本程序：

```
% chk_mysql_tables.sh /usr/local/mysql/data/sampdb
```

为了自动调用脚本程序，以一个cron作业设置它。如果你在系统上运行多个MySQL服务器，则可以多次运行chk\_mysql\_tables.sh，每次有不同的数据目录参数。

理想情况下，从chk\_mysql\_tables.sh中没有输出，不过，如果外部锁定禁止（这是MySQL 4及以后的版本中默认的），有可能MySQL服务器在你检查数据表时改变它。该脚本程序只检查

数据表，而不打算修复它，所以不会引起任何损坏。但是，`myisamchk`或`isamchk`工具程序或许会错误地报告数据表有问题，而实际上数据表正常。（这稍微有些遗憾，但是这比实际上有问题而工具程序报告没有损坏的反面影响要好。）如果你的系统支持外部锁定，该问题不会发生。参见第13.1节中有关外部锁定的内容。

还可以在机器引导期间，从系统启动的脚本程序中调用和运行`chk_mysql_tables.sh`脚本程序。如果使用BSD风格的系统，而且MySQL启动命令位于`/etc/rc.local`或等效地点，就可以从相同的文件中在启动MySQL服务器之前简单调用`chk_mysql_tables.sh`脚本程序。对于System-V风格的系统，在`/etc/rc.d`目录中查找。可以使用`rc.sysinit`和类似的脚本程序在MySQL服务器启动前运行`chk_mysql_tables.sh`。

### 13.3.2 使用备份恢复数据

恢复过程包括两个信息源——备份文件和二进制日志。备份文件使数据表恢复至备份执行时所处的状态。不过，数据表通常会在转储和发生问题的时间间隔内进行修改。二进制日志包括了用于制作这些修改的查询，所以可以把日志用做mysql的输入重复查询。（这就是为什么应当启用二进制日志。如果你还没有这样做，现在就应该这样做，并在读取之前产生一个新的备份。）

恢复过程是根据必须恢复多少信息而变化的。事实上，恢复整个数据库比恢复单个数据表容易，因为对数据库使用变更日志要比对数据表使用变更日志要容易一些。

#### 应该使用什么日志恢复操作

恢复操作中可以结合备份文件使用的日志是变更日志和二进制变更日志。二进制变更日志对恢复操作更有用，所以应该尽可能优先于变更日志而使用它。本小节的讨论是假定使用二进制日志而编写的。不过，二进制日志只可用在MySQL 3.23.14及以后的版本中。如果你没有这个功能，可以使用变更日志替代，并相应调整此项说明。

#### 1. 恢复整个数据库

首先，如果要恢复的数据库是包含权限表的mysql数据库，则必须使用`--skip-grant-tables`选项运行MySQL服务器。否则它会抱怨不能找到权限表。还有个好主意是使用`--skip-networking`，使得MySQL服务器在你执行恢复时拒绝所有远程连接的企图。当数据表恢复后关停MySQL服务器，并正常地重新启动MySQL服务器，以便使用权限表和正常地监听网络接口。

一般的恢复过程包括下列步骤：

- 1) 把数据库目录的内容拷贝至其他地方，或许以后要用到它们——或者执行已损坏数据表的事后检查分析。

- 2) 使用最新的备份文件重新装入数据库。如果使用`mysqldump`产生的备份作为包含SQL语句的文件，可使用它们作为mysql的输入。如果是使用从该数据库直接拷贝的文件（比如说，用`mysqlhotcopy`、`tar`或`cp`），则关停MySQL服务器，把文件直接拷贝至数据库目录，然后重新启动MySQL服务器（关停MySQL服务器的目的是不让MySQL服务器在拷贝操作期间试图访问该文件）。

3) 在制作备份后使用二进制变更日志对修改的数据库中的数据表重复进行查询。对于任何可使用的日志, 用mysqlbinlog把日志转换成ASCII格式, 并把其结果用做mysql的输入。指定--one-databases选项, 以便mysql只执行你所关心恢复的数据库查询。

如果你知道必须使用所有的日志文件, 通常可以使用它们位于所在目录中的下列命令:

```
% ls -t -r -l binlog.[0-9]* | xargs mysqlbinlog | mysql --one-database db_name
```

ls命令产生一个单列变更日志文件的清单, 按名字排序, 通常是按MySQL服务器产生它们的次序排列。不过, 如果文件名的数字扩展名不都具有同样的位数, 则不太可行。比如说, 如果你有日志名为binlog.998、binlog.999和binlog.1000的日志文件, ls对它们的排序将是binlog.1000位于第一。通过sort流水作业出的名称不予工作, 因为它以同样的方法给名字排序。有必要只根据扩展名的数值执行数字排序, 下列简短的perl脚本程序进行了这项工作:

```
#!/usr/bin/perl -w
# ext_num_sort.pl - sort filenames based on numeric extension value.

use strict;

my @files = <>;          # read all input lines
@files = sort {          # sort them by numeric extension
    my $anum = $1 if $a =~ /\.(\\d+)$/;
    my $bnum = $1 if $b =~ /\.(\\d+)$/;
    $anum <=> $bnum;
} @files;

print @files;            # print them
exit (0);
```

使用如下所示的脚本程序:

```
% ls -l binlog.[0-9]* | ext_num_sort.pl | xargs mysqlbinlog \
| mysql --one-database db_name
```

前面的讨论是假定你要使用所有的变更日志, 但是通常的情况是只需要使用其中一些——这些是在某些特定的备份以后写的。如果该日志是在备份时制作的, 并命名为binlog.1392、binlog.1393等等, 则可以按下面的叙述在其中重新运行这些语句:

```
% mysqlbinlog binlog.1392 | mysql --one-database db_name
% mysqlbinlog binlog.1393 | mysql --one-database db_name
...
```

如果你执行恢复和使用日志恢复信息的理由是因为有人随意地发出了DROP DATABASE、DROP TABLE或DELETE语句, 一定要在使用日志前从出现过这种语句的日志文件中删除那个语句! 为此, 把日志转换成ASCII格式, 并把它保存在一个文件中。然后编辑文件并把结果馈送给mysql:

```
% mysqlbinlog logfile > textfile
% vi textfile
% mysql --one-database db_name < textfile
```

如果没有二进制变更日志，但有文本变更日志，则不需要用mysqlbinlog，因为该日志已经是ASCII格式。在这种情况下，对给定的数据库从所有的日志中同时使用这些修改。如下所示：

```
% ls -l update.[0-9]* | ext_num_sort.pl | xargs cat \
| mysql --one-database db_name
```

为了使用各个日志，按照下列方式进行：

```
% mysql --one-database db_name < update.1392
% mysql --one-database db_name < update.1393
...
```

日志排序不正确的问题是由于扩展名的长度不同，这在以后是不会经常发生的。在某些版本（可能在4.1系列）中，MySQL服务器将改变成使用最少6位数的扩展名，而不是3位数。这种改变将使ext\_num\_sort.pl不再有用。

## 2. 恢复单个数据表

恢复各个数据表可能要比恢复一个数据库更难。如果你有mysqldump产生的备份文件，并且包含许多数据表用的数据，而不仅是你感兴趣的那一个，则你必须取出该文件的相关部分，并把它作为mysql的输入，这不过是最容易的一部分工作！难的部分是找出仅适用于这个数据表的日志的部分。可以借助于mysql\_find\_rows工具程序进行查找，它能从变更日志中或从已用mysqlbinlog转换成ASCII的二进制变更日志中取出多行进行查询。

另一种可能性是把整个数据库恢复到第二个空白的数据库中。再从那个数据库中使用mysqldump --add-drop-table转储要恢复的数据表，然后把它装回到原来的数据库中（--add-drop-table保证恢复操作是从取消的内容开始）。这个步骤实际上比试图恢复单个数据表更容易，只要从变更日志中取出相关各行就可以进行。另一种可能性（对于非BDB或InnoDB数据表）是把数据表文件从第二个数据库拷贝回原来数据库的数据库目录中。要保证把文件拷贝回数据库目录时，两个数据库用的MySQL服务器都关停。

## 3. 使用RESTORE TABLE恢复数据表

RESTORE TABLE是和BACKUP TABLE对应的，其工作是恢复使用BACKUP TABLE语句拷贝的MyISAM数据表。它可以用于MySQL 3.23.25及以后的版本中，并且需要FILE和INSERT权限。为了使用RESTORE TABLE，命名数据表或要恢复的数据表，以及备份文件位于MySQL服务器主机上的目录。假定以前使用BACKUP TABLE备份了三个数据表，如下所示：

```
BACKUP TABLE tbl1, tbl2, tbl3 TO '/archive/sampdb';
```

为了恢复这些数据表中的任意一个或全部，在RESTORE TABLE语句中命名需要的那些数据表，比如说，按下列所示恢复tbl1和tbl3：

```
RESTORE TABLE tbl1, tbl3 FROM '/archive/sampdb';
```

RESTORE TABLE把数据表的.frm文件和.MYD文件重新装入数据库目录，而且使用它们重新建立.MYI索引文件。

## 4. 恢复有外键关系的数据表

如果难于使用转储文件恢复有外键关系的数据表——因为这些数据表在文件中未按这些关

系所需的次序列出，则可以使用下列语句临时关闭键字检查：

```
SET FOREIGN_KEY_CHECKS = 0;
```

FOREIGN\_KEY\_CHECKS始见于MySQL 3.23.52版本。数据表导入后，重新启用键字检查：

```
SET FOREIGN_KEY_CHECKS = 1;
```

断开键字检查，允许你以任何次序建立和装入数据表，也可以加速装入。

必须关闭同一连接内的键字检查，该连接也用于重新装入转储文件，因为该设定仅影响当前的连接。用source命令装入文件就能实现这一点，而不是在命令行上命名它。假定有一个名叫dump.sql的导出文件，其装有来自名为mydata数据库的数据表，可如下所示装入它：

```
% mysql mydata
mysql> SET FOREIGN_KEY_CHECKS = 0;
mysql> source dump.sql;
mysql> SET FOREIGN_KEY_CHECKS = 1;
mysql> ...
```

第二个SET语句仅当转储文件装入后在mysql会话内打算发出更多语句时需要。如果在文件装入后退出，就没有必要了。

#### 5. 恢复InnoDB表空间或BDB数据表

当崩溃后MySQL服务器重新启动时，InnoDB数据表处理程序力求执行任何必要的自动恢复。不过，万一InnoDB检测出一个不可恢复的故障，启动将失败。在这种情况下，即便崩溃后InnoDB的恢复说不定失败，也设定innodb\_force\_recoveryMySQL服务器变量为非零值，在1~6之间，使MySQL服务器启动。为了设置该变量，把如下所示的一行放入[mysqld]选项组中：

```
set-variable = innodb_force_recovery=level
```

InnoDB处理程序尽量使用较低数值的保守策略。推荐的典型启动值level=4。MySQL服务器启动后用mysqldump转储InnoDB数据表，尽可能返回更多的信息，丢弃数据表，并从mysqldump输出文件中恢复它们。该过程会以内部一致的形式重新建立数据表，足以获得满意的恢复。执行恢复之后，从选项行中删除设定innodb\_force\_recovery这一行。

如果必须恢复整个InnoDB表空间，具体做法将取决于你当初是如何制作备份的（这里假定你使用第13.2.2节的“对InnoDB表空间或BDB数据表进行备份”小节中的说明已建立了一个备份）。

- 如果你使用的是直接拷贝方法，应该具有表空间文件、日志文件、每个数据表的.frm文件和确定InnoDB配置的MySQL服务器选项文件的拷贝。在MySQL服务器关停后，删除现有的InnoDB文件，并用备份拷贝替换它们。然后保证现有的MySQL服务器选项文件以与保存选项文件同样的方法列出InnoDB配置，并重新启动MySQL服务器。
- 如果运行mysqldump备份了InnoDB表空间，产生了一个SQL文件，其中含有从头重新建立数据表所必需的CREATE TABLE和INSERT语句，然后应该重新初始化表空间，并把转储文件重新装入表空间。随着MySQL服务器关停，抛弃任何现有的与InnoDB有关的文件：表空间文件（不是原始分区）、日志文件和所有InnoDB数据表的.frm文件。以最初进行的



同样方法重新配置表空间。(参见第11.5.5节的内容。保存的MySQL服务器选项文件的拷贝作为配置应该是什么样的记录是有帮助的。此外还要记住,如果使用任意的原始分区,初始化表空间是一个两步的过程。)当MySQL服务器结束建立新的表空间时,重新装入含有SQL语句的备份文件,以便使用它作为mysql的输入重新建立InnoDB数据表。

用备份恢复InnoDB表空间后,根据二进制日志把备份后又发生的数据操作都重新执行一遍。如果把表空间恢复作为整个数据库恢复的一部分,这是最容易的,因为在这种情况下能应用所有的修改。如果只恢复InnoDB表空间,应用该日志将比较复杂,因为你只是要使用InnoDB数据表所用的修改。

类似于InnoDB处理程序,崩溃发生后,BDB处理程序也会在你启动MySQL服务器时尝试进行自动恢复。如果启动过程因为不可恢复的BDB故障而失败,把所有的BDB日志文件从数据目录中移到其他目录中(或者当你不打算以后检查它们时就删除掉它们)。然后用--bdb-no-recover选项再次启动MySQL服务器。如果日志文件已损坏,或许MySQL服务器就能启动并新建一个BDB日志。如果MySQL服务器还是没有启动,就要用备份去替换替换BDB文件了:

- 如果你是直接拷贝有关文件,就应该有BDB数据表文件和BDB日志文件。关停MySQL服务器,从数据目录里把现有的BDB数据表和日志文件删掉,把它们替换为备份。
- 如果你使用的是mysqldump生成的备份,其中就会有重建各有关数据表的SQL语句。关停MySQL服务器,删除现有的BDB数据表和日志文件,重新启动MySQL服务器,最后通过把备份文件用做mysql程序的输入的办法加载之。

恢复完备份之后,根据二进制日志把备份后又发生的数据操作都重新执行一遍(要注意InnoDB恢复工作中提到的注意事项)。





## 第四部分 附录

附录A 获得并安装有关软件

附录B 数据列类型指南

附录C 操作符与函数用法指南

附录D SQL语法指南

附录E MySQL程序使用指南

附录F C API指南

附录G Perl DBI API指南

附录H PHP API指南

附录I 挑选ISP

A.2 获得MySQL及安装软件

MySQL软件是必不可缺少的；如果你的机器上还没有MySQL，那肯定得安装它。至于那些

## 附录A 获得并安装有关软件

本书里的示例都取材于一个名为sampdb的样板数据库，本附录将介绍如何获得sampdb发行版本。为了使用这个发行版本，还得让MySQL系统运转起来。因此，本附录还将介绍如何获得和安装MySQL及其相关软件，比如Perl DBI和CGI.pm模块、PHP、Apache等等。本附录内容覆盖了UNIX和Windows这两大类系统。不过，本附录的目的是把有关软件的安装办法集中汇总起来以方便大家的查阅和参考，而不是想取代各种软件中自带的安装指南。事实上，更希望大家去阅读那些指南。虽说本附录提供的信息应该能够满足大多数场合的需要，但软件自带的安装指南对大家在安装过程中遇到的问题提供了更有针对性的解决方案。以MySQL手册为例，其中有许多章节就是为了向各种平台上的MySQL安装问题提供解决方案而撰写的。

### A.1 获得样板数据库sampdb的发行版本

sampdb发行版本包含了建立和访问sampdb样板数据库所需要的各种文件。可以在下面网站上找到并下载它：

<http://www.kitebird.com/mysql-book/>

sampdb发行版本有tar压缩文件和ZIP压缩文件两种格式。tar格式的发行版本可以用下面两条命令之一来解压缩（如果你的tar命令不支持z选项，请使用第二条命令）：

```
% tar xzf sampdb.tar.gz
% gunzip < sampdb.tar.gz | tar xf -
```

ZIP格式的发行版本可以用WinZip、pkunzip或unzip等软件工具来解压缩。

在对sampdb发行版本进行解压缩的时候，它将创建一个名为sampdb的目录并在其中生成以下一些文件和子目录：

- 一个README文件，其内容是安装和使用sampdb发行版本的基本方法。这是应该阅读的第一个文件。发行版本的各个子目录里还包含一些内容更为具体的README文件。
- 一些用来建立和加载sampdb数据库的文件。这些文件可以对照着本书第1章中的有关内容来使用。
- 一个capi目录，其内容是第6章中的各种C语言示例程序。
- 一个perlapi目录，其内容是第7章中的各种Perl DBI脚本程序。
- 一个phpapi目录，其内容是第8章中的各种PHP脚本程序。

sampdb目录还有其他几个子目录，其中是一些在本书其他章节中提到的有关文件。

### A.2 获得MySQL及相关软件

MySQL软件是必不可少的；如果你的机器上还没有MySQL，那肯定得安装它。至于那些第



三方软件，只要把打算使用的那些安装上就行了。

- 如果想编写一些Perl脚本来访问MySQL数据库，就必须安装DBI和DBD::mysql模块。如果还打算编写基于Web的DBI脚本，那还得安装CGI.pm模块；当然还得有一个Web服务器才行。本书使用的Web服务器是Apache，但这并不表明其他Web服务器不能使用，大家可以随意选用。
- 如果想编写一些PHP脚本来访问MySQL数据库，就必须安装PHP。PHP主要用于编写基于Web的脚本，所以肯定还需要一个Web服务器。本书选用Apache服务器来配合PHP。

前面提到的这些软件大都有预编译好的二进制可执行代码版本。如果你使用的是一个Linux系统，就有各种各样的RPM文件可供选用。如果你喜欢从源代码开始自行编译有关软件，或者找不到适用于系统平台的二进制可执行代码版本，那还得准备一个C编译器（MySQL软件要用C++编译器来编译）。

如果你在某个ISP（Internet Service Provider，因特网服务提供商）处有账户，而这个ISP又提供有MySQL服务，那它很可能已经把以上这些软件都安装齐全了。如果是这种情况，就可以直接使用它们，而不必再研读本附录后面的内容了。如果不是这种情况，你就得自己去下载并安装有关软件了。下面是有关软件的官方发行站点，希望能对大家有所帮助。这些站点大都会有几个镜像站点，它们提供的软件是相同的，但因为地理距离的远近，有些站点的下载时间要比其他站点短得多。

软件包	官方站点网址
MySQL	<a href="http://www.mysql.com/">http://www.mysql.com/</a>
Perl模块	<a href="http://cpan.perl.org/">http://cpan.perl.org/</a>
PHP	<a href="http://www.php.net/">http://www.php.net/</a>
Apache	<a href="http://www.apache.org/">http://www.apache.org/</a>

应该根据自己的具体情况来挑选软件包的具体版本，考虑因素如下：

- 如果想得到最大限度的稳定性，就应该保守地选用软件包最新的稳定版本。这类版本不像开发版本那样有很多的试验性代码，它们既能提供最新的功能，又最大限度地修补了软件中的漏洞。如果不打算搞软件开发，这类版本应该是首选。
- 如果你喜欢为新事物冒一点险，或者所需要的功能只有最新的开发版本才能提供，就应该选择最新的开发版本。
- MySQL的预编译二进制可执行代码和PRM格式的发行版本都是用优化选项编译出来的，其效果往往要比用源代码版本里的配置脚本所能配置出来的效果要更好。MySQL开发组推荐人们尽量使用来源于[www.mysql.com](http://www.mysql.com)站点的二进制可执行代码版本。他们在编译MySQL软件时使用了一些商业化的优化编译器，从而使MySQL能够以更快的速度来运行。因此，这些发行版本里的程序通常要比你自行编译出来的代码执行得更快。此外，因为他们在各种编译器的使用方面有着丰富的经验，所以能够最大限度地避免或者绕开因编译器本身的漏洞而造成的缺陷代码，确保MySQL能够以最佳状态投入运行。

这些软件包的官方站点会告诉你哪些是最新的稳定版本、哪些是当前的开发版本。你可以利用这些站点为每个版本提供的功能改进清单来帮助自己挑选到最适合自己的版本。



如果你决定选用一个二进制可执行代码版本或者RPM格式的发行版本，发行版本的解压缩工作就相当于软件的安装工作，有关文件将解压缩到预定的目录里去，解压缩工作完成后，软件也就安装好了。你可能需要以root用户身份来对这些发行版本进行解压缩操作，因为有些文件需要解压缩到系统上的受保护的目录里去。

如果你决定选用一个源代码版本，应该先把它们解压缩到一个临时目录里，在那里完成各项配置和编译工作后，再把软件安装到最终的安装地点去。安装阶段的工作可能需要以root用户身份来进行，但配置和编译阶段的工作一般都用不着以root用户身份来进行。

如果你打算在UNIX系统上从源代码开始来安装有关软件，本附录介绍的软件包中有几个可以用configure工具来配置，这个工具可以大大简化很多种系统平台上的软件编译和安装工作。如果编译失败，就需要使用configure工具的另外一些选项重新来过。但在此之前，需要先把configure工具在你上次使用它时保存起来的选项和有关信息清除掉。可以用下面这条命令来清除configure工具以前保存的配置信息：

```
% make distclean
```

也可以使用下面这两条命令：

```
% rm config.cache
```

```
% make clean
```

#### 订阅邮件列表以寻求帮助

在准备安装一个软件包的时候，先订阅一份与该软件有关的邮件列表是一个很不错的注意：一旦遇到麻烦，你就可以迅速提出问题并得到别人的帮助。如果你准备安装的是一个开发版本，那就更应该订阅有关软件的邮件列表了：它可以让你随时掌握最新的漏洞报告和修补措施。即使你不想加入一个讨论组，也至少应该订阅它的通告表以便让自己能够在第一时间获得新版本的发布信息。邮件列表的订阅和使用办法可以在有关站点的主页上查到，软件包的发行站点通常也会提供这方面的信息。

### A.3 在UNIX系统上安装MySQL

市面上有MySQL不同版本的多种发行版本。目前，版本号在4.0系列的发行版本都很稳定，而版本号在4.1系列的发行版本则仍处于开发阶段。一般来说，在选定了版本类型（稳定或开发）及发行格式（二进制、RPM或源代码）之后，应该尽量挑选版本号最高的发行版本来使用。

MySQL发行版本的格式有二进制、RPM和源代码等几种。二进制码和RPM格式的发行版本比较容易安装，但你必须接受别人为该发行版本预先安排好的目录布局 and 默认配置。源代码格式的发行版本不太容易安装——因为你必须亲自去编译有关软件，但你对配置参数有着更多的控制权。比如说，既可以撇开服务器部分而只编译发行版本中的客户支持部分，也可以随心所欲地把软件安装到任何地方。

MySQL发行版本通常由以下一个或者多个组件构成：

- mysqld服务器。

- 客户程序 (mysql、mysqladmin等等) 和客户程序编程支持 (C语言开发库和各种头文件)。
- 文档。
- 性能评测数据库。
- 多语言支持。

源代码和二进制格式的发行版本通常都包含有上述全部组件。RPM文件往往只包含上述组件中的一部分, 因而需要安装多个RPM文件才能得到你想要的全部东西。

如果你只是需要连接到运行在另一台机器上的MySQL服务器去, 那就不需要安装服务器。但你几乎总是需要安装上客户端软件, 其理由如下:

- 如果你的机器上没有MySQL服务器, 就必须通过客户端软件才能连接到运行在另一台机器上的MySQL服务器去。
- 如果你的机器上有MySQL服务器, 也应该安装上客户软件。否则, 当你需要进行一些数据库管理或调试方面的工作时, 就将不得不去另外一台安装有客户端软件的机器上才行——这未免太麻烦了。
- 如果你想在MySQL上做一些软件开发方面的工作, 就需要使用一些API (Application Programming Interface, 应用程序设计接口)。但与MySQL开发工作有关的各种API大都要求机器上必须安装有MySQL的C语言客户程序开发库。比如说, 如果你想编写基于MySQL的Perl脚本, 就肯定要用到Perl DBI, 而Perl DBI又要求机器上必须安装有MySQL的C语言客户程序开发库才能正常工作。

### A.3.1 在UNIX系统上安装MySQL软件的基本步骤

在UNIX系统上安装MySQL软件的基本步骤如下:

- 1) 如果打算安装一个MySQL服务器, 就应该先创建一个系统登录账户, 好让MySQL服务器能够以这个账户下的用户和用户组身份运行。(只有首次安装需要进行这项工作; 如果是升级安装, 这一步骤可以省略。)
- 2) 获得并解压缩你想安装的MySQL发行版本。如果你打算从源代码开始安装, 就需要对下载到的软件包进行配置、编译和安装。
- 3) 运行mysql\_install\_db脚本程序, 对MySQL的数据目录和各种权限表进行初始化。(只有首次安装需要进行这项工作; 如果是升级安装, 这一步骤可以省略。)
- 4) 启动MySQL服务器。
- 5) 阅读本书第11章, 熟悉MySQL数据库系统的常规管理操作。你至少要掌握以下两项技能: 1) 如何启动和关闭MySQL服务器; 2) 如何用未授权的用户账户来运行MySQL服务器。

### A.3.2 为MySQL用户创建登录账户

只有首次安装或者在打算运行一个MySQL服务器的场合才必须进行这项工作; 如果是升级安装或者是只安装MySQL的客户软件, 这一步骤可以省略。

MySQL服务器允许你以UNIX系统上任何一种用户身份来运行, 但出于安全和管理方面的考虑, 不应该以根用户root身份来运行这个服务器。建议大家另外创建一个专用的系统登录账户

来管理和运行MySQL服务器，并把对MySQL数据目录进行维护、管理和调试等操作所需要的全部权限授予这个专用账户。请注意，用来创建新用户账户的命令在不同的UNIX系统上往往有着细微的差异，这方面的详细情况请参阅有关的系统文档。

在本书里，把MySQL的系统登录账户的用户名创建为mysqladm，把该账户的用户组创建为mysqlgrp。如果你安装的MySQL将只供你一个人使用，那完全可以沿袭你现有的登录账户信息——如果是这样的情况，应该把本书里的mysqladm和mysqlgrp分别替换为你自己的用户名和用户组名。如果你是利用RPM文件来安装MySQL软件，RPM安装过程将自动替你创建一个用户名为mysql的登录账户——如果是这样的情况，应该把本书里的mysqladm替换为mysql。

专门为MySQL创建一个未授权的用户账户而不使用root账户来运行MySQL服务器的好处是：

- 不使用root账户来运行MySQL的做法能够杜绝别人利用MySQL服务器作为安全漏洞而获得root账户的权限。
- 使用未授权的用户账户而不使用root账户来进行MySQL的维护和管理工作的做法更安全，即使出现意外失误，也不至于破坏整个系统。
- MySQL服务器创建的文件将由mysqladm用户而不是root用户拥有。而root用户拥有的文件越少，系统需要防范的隐患也就越少。
- 为MySQL单独创建一个专用账户的做法能够把各种MySQL活动条理清晰地局限在它自己的账户里，系统上发生的事情哪些与MySQL有关将一目了然。比如说，在用来存放各种crontab文件的目录里，对应于MySQL系统的mysqladm用户将单独拥有一个专用的文件。如果不是这样，MySQL系统的cron任务就将混杂在对应于整个系统的root用户的crontab文件里，让我们不容易看出哪些事情是需要由MySQL去定期执行的任务，哪些事情又是需要由root用户去定期执行的任务。

### A.3.3 获得并在UNIX系统上安装MySQL发行版本

在下面的讨论里，将用version来代表你准备安装的MySQL发行版本的版本号，用platform来代表你打算把MySQL软件安装在其中的系统平台的名称。为便于区分，MySQL发行版本里的文件名大都带有一个版本号和系统平台名称作为后缀。这里所说的“版本号”指的是诸如3.23.52或4.0.5-beta之类的东西，“系统平台名称”则指的是诸如sun-solaris2.8-sparc或pc-linux-gnu-i686之类的东西。

对于二进制格式的MySQL发行版本，如果文件名里有-max字样，就表明收录在该文件里的MySQL服务器能够提供一些附加功能，而这些附加功能又是那些标准的MySQL服务器所不能提供的。这些附加功能并不是一成不变的，所以你应该到MySQL的Web站点上去查清楚MySQL的标准发行版本与max发行版本到底有哪些最新的差异。如果你是从源代码开始来安装MySQL软件的，就可以在对源代码进行编译时利用configure脚本提供的各种选项来激活或者禁用这类附加的功能。

#### 1. 安装二进制发行版本

二进制格式的MySQL发行版本都有一个mysql-version-platform.tar.gz形式的文件名。首先，请根据版本号version和系统平台名称platform选择一个最适合具体情况的发行版本并把它下载到

准备用来安装MySQL软件的目录（比如/usr/local）里去。

然后，用下面两条命令之一来解压缩这个发行文件（如果你的tar命令不支持z选项，请使用第二条命令）：

```
% tar xzf mysql-version-platform.tar.gz
% gunzip < mysql-version-platform.tar.gz | tar xf -
```

发行文件的解压缩操作将自动创建一个名为mysql-version-platform的目录并把各有关文件放到事先安排好的地方去。这么长的目录名用起来当然很不方便，所以这里为它创建了一个名为mysql的符号链接：

```
% ln -s mysql-version-platform mysql
```

如果你打算把MySQL安装在/usr/local目录下，现在就可以用/usr/local/mysql来指称MySQL安装目录了。

如果不想在每次启动MySQL客户程序时都在命令行上敲入它完整的路径名，你还得修改系统上的环境变量PATH，把MySQL安装目录下的bin目录也包括在环境变量PATH的查找路径里。这个查找路径应该是在shell的某个启动文件里设置的。

如果只想安装和使用此MySQL发行版本所提供的客户程序而不需要运行其中的服务器，MySQL安装工作到这一步就算是完成了。现在，如果你是首次安装MySQL，那还需要根据第A.3.4节做一些收尾工作；如果你是在进行升级安装，可以直接跳到第A.3.5节。

## 2. 安装RPM发行版本

如果你使用的是Linux系统，就可以利用RPM文件来安装MySQL软件。根据具体情况，可能需要用到以下这些RPM文件：

- MySQL-client-version-platform.rpm——MySQL客户程序。
- MySQL-version-platform.rpm——MySQL服务器软件。
- MySQL-Max-version-platform.rpm——MySQL服务器功能扩展包，用来给标准的MySQL服务器增加一些额外的功能。在安装这个RPM文件之前，必须先安装好相应的MySQL-version-platform.rpm文件。
- MySQL-embedded-version-platform.rpm——嵌入式MySQL服务器libmysqld。
- MySQL-devel-version-platform.rpm——客户程序开发支持包（客户程序开发库和各种头文件），用来编写客户程序。如果你打算编写一些能够访问MySQL数据库的Perl DBI脚本，就必须安装这个RPM文件。
- MySQL-shared-version-platform.rpm——共享式客户程序开发库。
- MySQL-bench-version-platform.rpm——各种性能评测程序。这个RPM文件里的组件要求系统安装有Perl DBI支持。（请参阅第A.3.6节。）
- MySQL-version-src.rpm——服务器、客户以及各种性能评测程序的源代码。

RPM文件对其中各组件的安装位置已经做好了安排，所以可以在任何一个目录里来安装它们。给定一个RPM文件rpm\_file，可以用下面这条命令来查知有关组件将被安装到什么地方：

```
% rpm -qpl rpm_file
```



下面这条命令将安装RPM文件`rpm_file`（你可能需要以root用户身份来做这件事）：

```
# rpm -i rpm_file
```

因为MySQL的各种组件被划分到了不同的RPM文件里，所以通常需要安装多个RPM文件才能得到你想要的全部东西。下面这条命令将安装MySQL的客户支持：

```
# rpm -i MySQL-client-version-platform.rpm
```

服务器支持则要用下面这条命令来安装（如果你不需要用到Max服务器所提供的附加功能，可以省略第二条命令）：

```
# rpm -i MySQL-version-platform.rpm
# rpm -i MySQL-Max-version-platform.rpm
```

如果你需要利用MySQL客户程序开发支持来编写一些程序，就还得安装相应的RPM文件：

```
# rpm -i MySQL-devel-version-platform.rpm
```

如果你打算从源代码开始进行安装，可使用下面这条命令：

```
# rpm -recompile MySQL-version.src.rpm
```

如果不想在每次启动MySQL客户程序时都在命令行上敲入它完整的路径名，你还得修改系统上的环境变量PATH，把MySQL安装目录下的bin目录也包括在环境变量PATH的查找路径里。这个查找路径应该是在shell的某个启动文件里设置的。

如果只想安装和使用此MySQL发行版本所提供的客户程序而不需要运行其中的服务器，MySQL安装工作到这一步就算是完成了。现在，如果你是首次安装MySQL，那还需要根据第A.3.4节做一些收尾工作；如果你是在进行升级安装，可以直接跳到第A.3.5节。

### 3. 安装源代码发行版本

源代码格式的MySQL发行版本都有一个`mysql-version.tar.gz`形式的文件名，其中的`version`是MySQL软件的版本号。首先，把内容为MySQL源代码的压缩文件下载或复制到你想对它进行解压缩的目录里。然后，用下面两条命令之一来解压缩这个发行文件（如果你的tar命令不支持z选项，请使用第二条命令）：

```
% tar xzf mysql-version.tar.gz
% gunzip < mysql-version.tar.gz | tar xf -
```

发行文件的解压缩操作将自动创建一个名为`mysql-version`的目录并把各有关文件存放到事先安排好的地方去。用下面这条命令把当前路径切换到这个目录里：

```
% cd mysql-version
```

既然使用的是源代码格式的MySQL发行版本，就必须先对它进行配置和编译之后才能开始安装MySQL软件。如果你在上面几个步骤里遇到了麻烦，请到MySQL发行版本所附带的*MySQL Reference Manual*（MySQL参考手册）特别是与你的计算机型号相对应的有关章节里寻求解决方案。

现在，用configure命令来配置MySQL源代码发行版本：

```
% ./configure
```



可能需要在configure命令里安排一些配置选项。configure命令的配置选项可以用下面这条命令查出来：

```
% ./configure --help
```

下面是configure命令一些比较常用的配置选项：

- `--with-innodb`、`--without-innodb`——支持或者不支持InnoDB数据表的使用。在MySQL 4之前的版本里，必须明确地给出`--with-innodb`选项才能把InnoDB数据表处理程序包括在MySQL服务器里。在MySQL 4及以后的版本里，InnoDB机制将默认地被包括在MySQL服务器里，必须明确地给出`--without-innodb`选项才能排除有关组件。
- `--with-berkley-db`——支持BDB数据表的使用。
- `--without-server`——只建立客户端部分（客户程序或客户程序开发库）。如果你需要安装客户端组件以访问运行在另一台机器上的MySQL服务器，就应该加上这个选项。
- `--with-embedded-server`——安装嵌入式服务器库libmysqld。
- `--prefix=path_name`——在默认情况下，MySQL软件的根安装目录是/usr/local，它的数据目录、客户程序、服务器、客户程序开发库、头文件、使用手册页、多语言支持文件等组件将分别安装到该目录下的var、bin、libexec、lib、include、man、share等目录里。如果想为MySQL软件另行指定一个根安装目录，就需要使用`--prefix`选项。比如说，如果你给出了`--prefix=/usr/local/mysql`选项，MySQL软件的各种组件就都将安装到/usr/local/mysql目录里去。
- `--localstatedir=path_name`——这个选项用来改变数据目录的位置。如果不想把MySQL的数据目录放在它的根安装目录里，就可以利用这个选项来改变之。
- `--with-low-memory`——sql/sql\_yacc.cc源文件在编译时需要消耗大量的内存，这很容易导致编译工作半途而废，甚至在系统内存和交换空间（swap space）都比较充足的机器上也经常会出现问题。这类问题的故障现象通常是一条“Fatal Signal 11”（致命信号11）出错信息或者是系统中的虚拟内存被消耗殆尽。`--with-low-memory`选项能够降低编译器的内存使用量。

在运行完configure命令后，下面两条命令将分别完成编译和安装工作：

```
% make
% make install
```

如果没有在configure命令里用`--prefix`选项为MySQL指定一个你拥有其写权限的目录作为根安装目录，就可能需要以root用户身份来执行make install命令。

如果不想在每次启动MySQL客户程序时都在命令行上敲入它完整的路径名，你还得修改系统上的环境变量PATH，把MySQL安装目录下的bin目录也包括在环境变量PATH的查找路径里。这个查找路径应该是在shell的某个启动文件里设置的。

如果只想安装和使用此MySQL发行版本所提供的客户程序而不需要运行其中的服务器，MySQL安装工作到这一步就算是完成了。现在，如果你是首次安装MySQL，那还需要根据第A.3.4节做一些收尾工作；如果你是在进行升级安装，可以直接跳到第A.3.5节。

### A.3.4 初始化数据目录和权限表

刚安装好的MySQL软件还不能立刻投入使用，还得先对mysql数据库进行初始化才行。在mysql数据库中的各种权限表里，存放着MySQL服务器的各种访问权限控制信息。不过，只有首次安装或者在打算运行一个MySQL服务器的场合才必须进行这项工作；如果是升级安装或者是只安装MySQL的客户端软件，这一步骤可以省略。

在下面的讨论内容里，将用DATADIR来代表数据目录的路径名。一般来说，本小节里的命令都需要以root用户身份才能执行。但如果你是通过系统分配给MySQL的登录账户进入系统的（即mysqladm用户）或者你把MySQL安装在了自己的账户下（即安装的MySQL只供你一个人使用），那你即便不是root用户也能执行这些命令——如果是这种情况，你将用不着执行下面的chown和chmod命令。

要想对数据目录、mysql数据库和默认的权限表进行初始化，只需把当前路径切换到MySQL的根安装目录再执行mysql\_install\_db脚本就行了。（如果你当初是使用RPM文件来安装MySQL软件的，就不需要这一步骤；因为安装过程已经自动执行过mysql\_install\_db脚本了。）比如说，如果你把MySQL软件安装到了/usr/local/mysql目录里，就需要使用下面两条命令：

```
# cd /usr/local/mysql
# ./bin/mysql_install_db
```

如果mysql\_install\_db脚本执行失败，请参考MySQL Reference Manual（MySQL参考手册）对MySQL安装过程的讨论去查找问题原因和解决方案。但有一点需要大家特别注意：如果mysql\_install\_db脚本没有成功地执行到结束，它创建出来的各种权限表就很可能是不完整的。但在第二次执行的时候，如果mysql\_install\_db脚本发现这些权限表已经存在，就不会再去重新创建它们。因此，在第二次执行mysql\_install\_db脚本之前，必须把它在第一次执行时创建出来的权限表全都删除掉。下面这条命令可以把整个mysql数据库删除掉：

```
# rm -rf DATADIR/mysql
```

在成功地运行完mysql\_install\_db脚本之后，别忘了用下面两条命令对数据目录里的所有文件的属主信息（用户和用户组）和访问模式做出相应的修改。假定用户和用户组是mysqladm和mysqlgrp，则命令如下：

```
# chown -R mysqladm.mysqlgrp DATADIR
# chmod -R go-rwx DATADIR
```

先用chown命令把文件的属主（owner，也叫拥有者）全部修改为你专为MySQL而创建的那个系统登录用户（在我们的例子里，即mysqladm.mysqlgrp），再用chmod命令把文件的访问模式全部设置为只允许用户mysqladm进行访问（读、写、执行）而拒绝其他任何用户进行访问的样子。

### A.3.5 启动MySQL服务器

只有必须运行有一个MySQL服务器的场合才需要进行这一步骤。如果你只安装了MySQL软件的客户程序部分，可以跳过这一小节。本小节里的命令都需要在MySQL的根安装目录里执行

(就像上一小节里的命令一样)。一般来说,本小节里的命令都需要以root用户身份才能执行。但如果你是通过系统分配给MySQL的登录账户进入系统的(即mysqladm用户)或者你把MySQL安装在了自己的账户下(即安装的MySQL只供你一个人使用),那你即便不是root用户也能执行这些命令——如果是这种情况,请省略以下命令中的--user选项。

下面的第一条命令把当前路径切换到MySQL的根安装目录(以/usr/local/mysql为例),第二条命令启动了MySQL服务器:

```
# cd /usr/local/mysql
# ./bin/mysqld_safe --user=mysqladm &
```

--user选项的作用是让服务器以mysqladm用户来运行。(注意:mysqld\_safe脚本在MySQL 4之前的版本里的名字是safe\_mysqld。)

在安装过程的这一阶段,你可能还得做以下几件事:

- MySQL软件的默认安装允许MySQL系统的root用户不使用口令来进行连接。为安全起见,最好趁现在给它设置一个口令。
- 可以把MySQL安排为整个系统的开、关机过程的一个组成部分,让它在系统开机时自动开始运行,在系统关机时自动结束运行。
- 可以把--user选项放到MySQL的某个选项文件里,这样,你就不必在每次启动MySQL的时候都不得不指定它了。
- 有选择地激活MySQL的各种日志功能。这对服务器的监视工作和数据恢复工作都有好处。
- 如果你的服务器支持使用InnoDB数据表,应该趁现在这个机会对InnoDB数据表处理程序进行配置。

以上几种操作的具体步骤可以在本书的第11章查到。

### A.3.6 在UNIX系统上安装Perl DBI支持

如果你想编写一些能够访问MySQL数据库的Perl脚本,就需要安装DBI软件。需要安装两组DBI模块,一组是负责提供各种DBI基本驱动程序的DBI模块,另一组是负责提供各种MySQL专用驱动程序的DBD::mysql模块。DBI需要Perl 5.005\_003或更高的版本才能正常工作。(如果你的系统还没有安装Perl,请从<http://www.perl.com/>站点下载一个Perl发行版本并在安装DBI支持之前把它安装好。)系统还必须安装有MySQL的C语言客户程序开发库,因为DBD::mysql模块需要用到它。(作为MySQL软件的组件之一,这个C语言程序开发库应该已经安装好了。)如果你还想编写一些基于Web的DBI脚本,那就还得安装CGI.pm模块。

可以利用perldoc命令来查知某个Perl模块是否已经安装好了;如果已经安装了该模块,perldoc命令就会把该模块的文档显示出来,如下所示:

```
% perldoc DBI
% perldoc DBD::mysql
% perldoc CGI
```

如果想把Perl模块安装到UNIX系统上,最简单的办法是使用CPAN shell。先以root用户身份登录,然后发出以下命令:

```
# perl -MCPAN -s shell
cpan> install DBI
cpan> install DBD::mysql
cpan> install CGI
```

还可以从cpan.perl.org站点下载tar压缩文件形式的源代码发行版本。假设你下载了一个名为dist\_file.tar.gz的压缩文件。首先,用下面两条命令之一来解压缩这个发行文件(如果你的tar命令不支持z选项,请使用第二条命令):

```
% tar xzf dist_file.tar.gz
% gunzip < dist_file.tar.gz | tar xf -
```

然后,把当前路径切换到tar命令创建的发行文件目录,再执行下面的命令(可能需要以root用户身份来执行make install命令):

```
% perl Makefile.PL
% make
% make test
# make install
```

不管你选择的是哪一种安装办法,在安装DBD::mysql的时候,必须回答Perl提出的以下几个问题:

- Which drivers do you want to install (你想安装哪些驱动程序)?  
可以选择MySQL和mSQL的任意组合。但除非你还打算运行mSQL,否则还是简单地只选择MySQL好了。
- Do you want to install the MysqlPerl emulation (你想安装MySQLPerl仿真接口吗)?  
MysqlPerl是老式的Perl语言MySQL编程接口,现在已经没有什么人使用它了。如果你手里没有老式的MysqlPerl脚本,也就用不着激活DBI模块中的MysqlPerl仿真支持功能——请选择“No”。
- Where is your MySQL installed (你的MySQL软件安装在哪里)?  
这个问题的答案是MySQL安装根目录;更准确地讲,它应该是你用来存放MySQL头文件的那个目录的“爷”目录。如果你没有把MySQL安装到其他不标准的位置,这个安装地点就应该是/usr/local或/usr/local/mysql。
- Which database should I use for testing the MySQL drivers (我要用哪个数据库来测试MySQL驱动程序)?  
这个问题的默认答案是test数据库,如果你没有把MySQL权限表里的匿名用户项删掉的话,就接受这个答案好了。但如果你真的已经把匿名用户项删掉了的话,就需要另行指定一个数据库(它必须是你有权去访问的),并在最后的两个问题里给出一个合法的MySQL用户名和口令。
- On which host is the database running (数据库运行在哪个主机上)?  
这个问题里的“数据库”就是前一个问题里的“数据库”。如果在你的机器上就有一个MySQL服务器,选择localhost就足以解决问题。如果不是这样,就需要另行指定一个运行有MySQL服务器(它必须是你有权去访问的)的主机名。而且,当你发出make test命令



的时候，另行指定的那台主机上的MySQL服务器必须是运行着的；否则，测试工作将会失败。

- Username for connecting to the database? Password for connecting to the database (用来连接数据库的用户名和口令是什么)？

这两个问题里的“数据库”就是前两个问题里的“数据库”。默认的用户名和口令都是undef——这将使有关的驱动程序以匿名用户进行连接。如果你需要以非匿名用户来进行连接，请另行给出一组合法的用户名和口令。

如果在安装Perl模块的时候遇到了麻烦，可以到相关发行版本的README文件或者相关DBI邮件列表的FAQs（常见问题解答）里去寻找解决方案。安装过程中的大多数问题都能在那里找到解答。

### A.3.7 在UNIX系统上安装Apache和PHP

下面的讨论假设你打算把PHP运行作为Apache服务器httpd的一个DSO(dynamic shared object, 动态共享对象)模块。这就意味着你必须先安装好Apache服务器，然后再去配置和安装PHP。如果你的系统还没有安装好Apache，就需要先安装它。可以找一个二进制格式的Apache发行版本（它的DSO支持功能必须被激活）来直接进行安装，也可以找一个源代码格式的Apache发行版本来先编译（编译时要激活其DSO支持功能）再安装。安装好Apache之后，再用下面两条命令之一来配置PHP发行版本：

```
% ./configure --with-mysql --with-apxs
% ./configure --with-mysql --with-apxs=/path/to/apxs
```

--with-mysql选项的作用是把MySQL客户端支持包括在所建立的PHP模块里——如果没有加上这个选项，基于MySQL的PHP脚本将无法正常工作。如果第一条命令因configure命令没有找到apxs脚本而执行失败，请使用明确地给出了apxs脚本存放路径的第二条命令（辅助脚本apxs的正式名称是Apache Extension Tool（Apache扩展工具），它的用途是把Apache服务器的配置信息提供给其他模块）。配置好PHP之后，用下面这些命令来编译和安装它（可能需要以root用户身份来执行make install命令）：

```
% make
# make install
# cp php.ini-dist /usr/local/lib/php.ini
```

cp命令的作用是把基准的PHP初始化文件安装到PHP能找到的地方去。如果你愿意，也可以用php.ini-recommended来代替php.ini-dist——最好先对照一下这两个初始化文件，然后再挑出一个你最喜欢的来用。

安装好PHP之后，下一步是编辑Apache的配置文件httpd.conf——要告诉Apache两件事：1）在启动时要加载上PHP模块；2）如何识别和运行PHP脚本。

我们先来说第一件事。要想让Apache在启动时加载上PHP模块，就必须在httpd.conf文件中与PHP有关的选项设置段里加上相应的LoadModule和AddModule指令（应该先在它的配置文件里找一找，看有没有与下面两条指令类似的东西）。这些指令应该已经在PHP模块的安装过程中



被添加好了。如果不是这样，就需要由你来做这件事。这两条指令应该是这样的：

```
LoadModule php4_module libexec/libphp4.so
AddModule mod_php4.c
```

接下来再说第二件事——编辑httpd.conf文件以便让Apache知道如何识别和运行PHP脚本。Apache是通过文件名后缀来识别各种脚本的，PHP脚本当然也不例外。PHP脚本最常见的文件名后缀是.php（本书示例中使用的正是它），另一个常见的文件名后缀是.phtml。为以后着想，还是把这两个文件名后缀都告诉给Apache为好——即使你本人总是使用.php作为PHP脚本的文件名后缀，但谁敢保证你今后不会用到别人写的、以.phtml为文件名后缀的PHP脚本呢？于是，我们在Apache服务器的配置文件httpd.conf里添加了这样两行文字：

```
AddType application/x-httpd-php3 .php
AddType application/x-httpd-php3 .phtml
```

还应该告诉Apache这样一件事：如果某个URL的末尾没有给出一个文件名，就把相关目录里的index.php或index.phtml文件识别为该目录的默认主页文件。httpd.conf文件里应该有一行类似于下面这样的文字：

```
DirectoryIndex index.html
```

请把它修改为：

```
DirectoryIndex index.php index.phtml index.html
```

编辑好Apache的配置文件之后，如果系统上的httpd服务器正在运行，请把它停下来，然后再重新启动之。在大多数系统上，下面两条命令可以完成这一工作（这两条命令通常都要求你必须是root用户才能执行）：

```
# /usr/local/apache/bin/apachectl stop
# /usr/local/apache/bin/apachectl start
```

你还可以把Apache配置成在系统开机时自动启动运行、在系统关机时自动结束运行；有关的具体配置步骤可以在Apache的文档里查到。一般说来，这需要你安排系统在开机时自动运行一个apachectl start命令，等关机时再自动运行一个apachectl stop命令。

如果在安装PHP的时候遇到了麻烦，可以到PHP发行版本自带的INSTALL文件中的“VERBOSE INSTALL”部分去寻找解决方案。（建议大家抽时间好好读读这个文件，其中有很多非常有用的信息。）

## A.4 在Windows系统上安装MySQL

MySQL可以运行在Windows NT / 2000 / XP等基于NT的系统上，也可以运行在Windows 95 / 98 / Me等非NT系统上。为了做到这一点，机器必须安装有TCP/IP支持。如果你使用的是Windows NT 4，就必须先把系统升级到Service Pack 3或更高的版本才行。如果你使用的是Windows 95，就必须先把Winsock软件升级到2或更高的版本才行。

如有可能，建议大家最好是使用基于NT的Windows版本。这样，就能把MySQL服务器安排为一项服务并让Windows在系统开机/关机的时候自动去启动/停止这项服务的运行了。NT系统

上的MySQL服务器还支持客户（程序）通过命名管道来建立连接。

除MySQL服务器和客户程序以外，你可能还需要安装针对MySQL的ODBC（Open Database Connectivity，开放数据库连接，由微软公司研发）驱动程序MyODBC——有了MyODBC，那些符合ODBC标准的程序就能直接访问MySQL数据库。比如说，如果你安装了MyODBC，就可以用Microsoft Access等ODBC程序来连接MySQL服务器了。

适用于Windows系统的MySQL发行版本可以在MySQL的Web站点上找到，它们通常是一些文件名形式为mysql-version-win.zip的ZIP文件。如果你想解压缩一个ZIP文件，可以先双击一下ZIP文件试试。如果这一招不管用，就得使用WinZip、pkunzip或unzip程序了。ZIP文件里的内容将被解压缩到一个文件夹里去，这个文件夹里会有一个Setup.exe文件，执行这个Setup.exe文件将开始安装MySQL软件。MySQL软件的默认安装位置是C:\mysql目录，但完全可以另行指定一个不同的目录来安装MySQL软件。

在Windows系统上安装好MySQL软件之后，通常用不着再去对数据目录或各种权限表进行初始化，因为这些事情都已经在发行版本里提前安排好了。但如果你把MySQL软件安装到一个不是其默认安装目录C:\mysql的其他地方去了，那么，为了确保MySQL服务器能够找到其安装基目录和数据目录的准确位置，就必须在它启动时肯定会读取的某个MySQL选项文件里增加一个[mysqld]选项组。（选项文件可以是根目录里的C:\my.cnf文件或Windows系统目录里的my.ini文件。）比如说，假设你把MySQL安装到了E:\mysql目录里，那么，与之对应的选项组就应该包括以下内容（注意，路径名里使用的是斜杠字符“/”而不是反斜杠字符“\”）：

```
[mysqld]
basedir = E:/mysql
datadir = E:/mysql/data
```

需要提醒大家注意的是，如果你真的把MySQL安装到一个不是其默认安装目录C:\mysql的其他地方去了，请不要忘记对后面内容里的某些命令的路径名做相应的修改。

在Windows系统上，有以下几种MySQL服务器可供选用：

服 务 器	说 明
mysqld	标准服务器
mysqld-nt	经过优化的服务器，支持命名管道
mysqld-max	经过优化的服务器，支持事务处理和符号链接
mysqld-max-nt	类似于mysqld-max，但支持命名管道

当运行在基于NT的系统上时，名字里有-nt字样的服务器将支持客户（程序）通过命名管道来建立连接。在MySQL 3.23.50之前的版本里，对命名管道的支持是被默认激活的；但在MySQL 3.23.50及以后的版本里，如果你想使用命名管道来建立连接，就必须在MySQL选项文件中的[mysqld]组里明确地加上下面这一行文字：

```
[mysqld]
enable-named-pipe
```

在基于Windows NT的系统上，mysqld-nt服务器还可以被安装为一项随Windows启动而自动启动的服务（你也可以把下面这条命令中的mysqld-nt替换为mysqld-max-nt）：

```
C:\> C:\mysql\bin\mysqld-nt --install
```

如果用--install-manual选项来代替上面命令里的--install选项，服务器就将被安装为一项需要由你来手动启动的服务，不再随Windows启动而自动启动。

把mysqld-nt服务器安装为一项服务之后，如果还需要用到它的其他选项，可以把它们放到MySQL选项文件中的[mysqld]组里去。

把MySQL服务器安装为一项服务之后，我们就可以利用Windows的服务管理来手工启动它。Windows服务管理器可以在Windows的控制面板里找到：它或者是控制面板窗口里一个名为Service（服务）的图标，或者是控制面板中的管理工具（Administrative Tools）子窗口里的一个项目。我们还可以用下面这条命令来启动这项服务：

```
C:\> net start MySQL
```

终止这项服务的办法也有好几个，既可以使用Windows的服务管理器，也可以使用下面两条命令之一：

```
C:\> net stop MySQL
```

```
C:\> C:\mysql\bin\mysqladmin -u root shutdown
```

如果你想删除作为服务运行的MySQL服务器，需要先终止服务器的运行（如果它正在运行的话），然后再发出下面这条命令：

```
C:\> C:\mysql\bin\mysqld-nt --remove
```

为避免Windows的服务管理器与你在DOS提示符下发出的命令发生冲突，在准备从DOS提示符发出与服务有关的命令之前，最好先把Windows的服务管理器关掉。

对于非NT系统（或者虽然是NT系统，但你却没有把MySQL服务器安装为一项服务的场合），必须以手动方式通过命令行来启动或者终止MySQL服务器的运行。下面这条命令将启动mysqld服务器（也可以把下面这条命令中的mysqld替换为mysqld-max）：

```
C:\> C:\mysql\bin\mysqld
```

如果你愿意，还可以在命令行上同时设定其他的选项。终止这个服务器要使用mysqladmin客户程序，如下所示：

```
C:\> C:\mysql\bin\mysqladmin -u root shutdown
```

如果你想让MySQL服务器运行在控制台模式（即让出错信息显示在一个控制台窗口里），请用下面这条命令来启动它：

```
C:\> C:\mysql\bin\mysqld --console
```

任何一种MySQL服务器都可以运行在控制台模式下。此时，需要把其他选项写在命令行上的--console选项的后面或者把它们放到MySQL的某个选项文件里去。终止一个运行在控制台模式下的服务器要使用mysqladmin客户程序。

如果你在启动MySQL服务器的时候遇到了麻烦，请查阅MySQL Reference Manual（MySQL参考手册）中与Windows安装有关的内容。

如果不想在每次启动MySQL客户程序时都在命令行上敲入它完整的路径名，还得修改你系

统上的环境变量PATH，把C:\mysql\bin目录也包括在环境变量PATH的查找路径里。（如果你把MySQL安装到了另外的地方，请对环境变量PATH中的路径名做相应的调整。）环境变量PATH通常是在AUTOEXEC.BAT文件里设置的；如果你使用的是一个基于NT的系统，还可以通过Windows控制面板里的System（系统）项来设置这个路径。

注意，MySQL软件的默认安装没有给MySQL系统的root用户设置口令，任何人都能以root用户身份去连接服务器而不需给出口令。这是一个很大的安全隐患，所以应该尽快给MySQL的root用户设置口令，具体操作步骤见本书第11章。第11章还介绍了其他一些操作，比如激活日志功能或配置InnoDB数据表处理程序等。

#### A.4.1 在Windows系统上安装Perl DBI支持

如果想在Windows系统上安装Perl模块，最简单的办法是先从www.activestate.com站点下载一份ActiveState Perl发行版本并把它安装好，然后再去安装所需要的其他Perl模块。ppm（Perl Package Manager，Perl软件包管理器）程序是专为这一目的而开发的工具。如果你想知道自己的系统上已经安装了哪些Perl模块，可以使用下面这条命令：

```
C:\> ppm info
```

下面这些命令将把相应的Perl模块安装到你的系统上。CGI.pm模块很可能已经被安装在系统上了，但另两个模块还是要由你来安装。

```
C:\> ppm
ppm> install DBI
ppm> install DBD:mysql
ppm> install CGI
```

#### A.4.2 在Windows系统上安装Apache和PHP

二进制格式的Apache和PHP发行版本可以分别在本附录开头部分列出的它们的官方站点上找到。在Apache 1.3.x版本下，PHP只能运行为一个独立的CGI程序。但在Apache 2.x版本下，PHP既可以运行为一个独立的程序，也可以运行为一个Apache模块。

#### A.4.3 在Windows系统上安装MyODBC

MyODBC 3的各种发行版本通常被打包为一个文件名形式是MyODBC-version.exe的可执行文件。只需下载并执行这个文件就能把MyODBC安装好。MyODBC 3之前的版本通常被打包为ZIP文件。NT和非NT系统有着不同的发行版本（可以根据文件名里的-nt和-win95来区分它们）。如果你想安装ZIP格式的发行版本，先双击它的文件名进行解压缩，然后再运行结果文件夹里的Setup.exe程序。如果你在用Setup.exe程序安装MyODBC的时候看到一条出错信息说“Problems while copying MFC30.DLL”（复制MFC30.DLL时发生问题），就说明有其他程序正在使用MFC30.DLL文件。此时，可以试着选择一下出错信息窗口里的Ignore（忽略）选项。MyODBC应该能够完成安装并不受出错信息的影响正常工作。如果这一招不奏效，请把

Windows重新启动到安全模式后再运行一次Setup.exe程序。

在安装好MyODBC的发行版本之后，还得通过Windows控制面板里的ODBC项目对有关的驱动程序进行配置。如果在Windows控制面板里没有找到ODBC项目，那它可能是被放到控制面板中的管理工具（Administrative Tools）子窗口里去了，其名字也可能会变成Data Sources (ODBC)（数据源(ODBC)）或ODBC Data Sources（ODBC数据源）。运行ODBC项目，你将看到一个用来设置DSN（data source name，数据源名）的窗口。单击选中User DSN（用户DSN）标签页，再单击Add（添加）按钮打开一个新窗口，新窗口里列出了各种可供选用的数据源驱动程序。在列表里选中MySQL驱动程序，然后单击Finish（完成）按钮，屏幕上将出现一个新窗口供你输入该数据源的各种连接参数。输入有关的连接参数，然后单击OK（确定）。

下面是一个输入有关连接参数的参考示例，我们将以用户名sampadm和口令secret连接到本地主机中的sampdb数据库：

字段名称	含义及取值
Data Source Name:	数据源: sampdb-dsn
Host Name (or IP):	主机名: localhost
Database Name:	数据库名: sampdb
User:	用户名: sampadm
Password:	口令: secret
Port:	端口号: 3306

配置好MyODBC之后，就可以使用遵守ODBC标准的其他程序来访问MySQL数据库了。我们来看一个例子。MyODBC的常见用途是从Microsoft Access去连接一个MySQL服务器。在安装并配置好MyODBC之后，如果你想在Access里连接MySQL服务器，请按以下步骤操作：

- 1) 启动Access程序。
- 2) 打开一个现有的数据库或者创建一个新的数据库。
- 3) 在File（文件）菜单里，选择Get External Data，再选择Link Tables。
- 4) 在新出现的窗口里，单击Type（类型）菜单中的Files（文件），然后选中ODBC Databases（ODBC数据库）。
- 5) 选中前面在控制面板中的ODBC项目里配置的DSN，开始连接MySQL。
- 6) 选择你想使用的MySQL数据表。

经过以上步骤之后，就可以通过Access来使用所选中的数据表了。



## 附录B 数据列类型指南

本附录对MySQL提供的各种数据列类型做了详细的介绍。各种数据列类型的使用方法在本书第2章里有详细的讨论。在本附录里列出的数据列类型至少从MySQL 3.22.0版本开始就已经存在了。但从那时起，有些类型的行为已经发生了改变，我们将在本附录的有关条目里对这类情况加以注明。

在本附录中的类型名定义里，我们使用以下约定：

- **方括号 ([ ])**——可选信息。
- **M**——最大显示宽度；如无特殊说明，*M*应该是从1 ~ 255之间的一个整数。
- **D**——小数点后面的数字位数。*D*应该是从1 ~ 30之间的一个整数。同时，*D*不得大于 *M*-2；否则，*M*值将被调整为*D*+2。

*M*和*D*相当于ODBC术语中的“显示范围”和“精确度”。

在介绍每一种数据列类型的时候，我们将依次给出以下几方面的信息：

- **含义**——对该类型的一个简短描述。
- **可用属性**——能够通过CREATE TABLE或ALTER TABLE语句有选择地加在数据列类型上的属性关键字。这些属性将按字母表顺序依次列出，但这不表示你在CREATE TABLE或ALTER TABLE语句里也必须按同样的顺序来列出这些属性。CREATE TABLE和ALTER TABLE语句的语法请参阅附录D。本附录各数据列类型条目里列出来的属性是将要介绍的全局属性的补充。
- **最大长度**——适用于字符串类型，它指的是该类型的数据列值被允许达到的最大长度。
- **表示范围**——适用于数值类型以及日期和时间类型，它指的是该类型所能表示的取值范围。整数类型因为存在带正负符号和不带正负符号两种情况，所以有两个表示范围；不同情况有不同的表示范围。
- **零值**——日期和时间类型的“零”值指的是MySQL在你试图把一个非法值插入到该类型的数据列里时真正放入有关数据列里的值。
- **默认值**——类型定义没有明确地设定DEFAULT属性时的默认值。
- **存储空间占用量**——存储有关类型的数值所需要使用的字节数。有些类型的存储空间占用量是一个固定的数字，有些类型的存储空间占用量却是一个可变的数字，要由具体存入有关数据列里的数值的长度来决定。
- **比较方式**——适用于字符串类型，用来表明该类型上的比较操作是否需要区分字母的大小写情况。这对排序和索引操作也有影响，因为这两种操作都是在比较操作的基础上进行的。
- **同义词**——类型名称的同义词，即同一类型的不同名称。
- **其他事项**——有关类型需要注意的其他方面。

有些全局属性适用于全部或者几乎全部的数据列类型。所以我们把它们列在下面，在有关条目里就不逐一列出了：

- 属性NULL和NOT NULL可以用在任何一种数据列类型上。
- 属性DEFAULT *default\_value*可以用在几乎全部的数据列类型上。默认值必须是常数。比如说，当你声明一个DATETIME数据列的时候，不能使用DEFAULT NOW()来为它设定默认值。此外，为TIMESTAMP数据列或者带有AUTO\_INCREMENT属性的数据列设定的默认值将是无效的，而为BLOB或TEXT数据列设定默认值的做法则是非法的。（根据计划，未来的MySQL 4.1系列版本将支持使用不是常数的默认值和为BLOB或TEXT数据列设定默认值的做法，但这两个目标在本书脱稿时仍未实现。）

## B.1 数值类型

MySQL的数值类数据列类型包括整数和浮点数两大类，可以根据自己需要表示的数值范围来进行选用。

如果设定了ZEROFILL属性，数值类型的值将用前导的零补足到有关数据列的显示宽度。

只要给某个整数类型的数据列加上了AUTO\_INCREMENT属性，就必须把它同时声明为一个PRIMARY KEY或一个UNIQUE索引。试图把NULL值插入一个AUTO\_INCREMENT数据列的做法将导致MySQL把下一个序列编号值（这个新编号的大小通常就等于该数据列的当前最大值再加上一个1）自动插入数据列。有关AUTO\_INCREMENT数据列的详细讨论请参见第2章。

如果数值类型的数据列带有UNSIGNED属性，就不允许负值出现在其中；但MySQL 4.0.2之前的FLOAT和DOUBLE数据列不支持UNSIGNED属性。（事实上，MySQL 4.0.2之前的版本允许你在声明FLOAT或DOUBLE数据列的时候给它加上UNSIGNED属性，但并不能有效地阻止负值被存入这些数据列。如果这些数据列还同时被设定了ZEROFILL属性的话，就很可能导致负值无法被正确显示的现象。）

在某些场合，即使你只设定了一种属性，也会导致其他几种属性也被激活。给数值类型设定上ZEROFILL属性的做法将使有关数据列被自动地设定为UNSIGNED。从MySQL 3.23开始，只要设定了AUTO\_INCREMENT属性，有关数据列就将被自动设定为NOT NULL。

还有一点需要大家特别注意：虽然DESCRIBE和SHOW COLUMNS命令会报告说某个AUTO\_INCREMENT数据列的默认值是NULL，但你却无法真的把一个NULL值插入到这个数据列里去——它只表明这样一个事实：如果你在创建一条新记录时给AUTO\_INCREMENT数据列赋值了一个NULL，就会在该数据列里“制造”出一个默认值（即下一个序列编号）来。

### TINYINT [(M)]

含义：一个非常小的整数。

可用属性：AUTO\_INCREMENT、UNSIGNED、ZEROFILL。

表示范围：带符号：-128~127 ( $-2^7 \sim 2^7 - 1$ )；无符号：0~255 ( $0 \sim 2^8 - 1$ )。

默认值：如果数据列允许使用NULL值，默认值为NULL；如果带NOT NULL属性，默认值为0。

**存储空间占用量：**1个字节。

**同义词：**INT1 [(M)]。此外，BIT和BOOL是TINYINT(1)的同义词。

## SMALLINT [(M)]

**含义：**一个小整数。

**可用属性：**AUTO\_INCREMENT、UNSIGNED、ZEROFILL。

**表示范围：**带符号： $-32\,768 \sim 32\,767$  ( $-2^{15} \sim 2^{15}-1$ )；无符号： $0 \sim 65\,535$  ( $0 \sim 2^{16}-1$ )。

**默认值：**如果数据列允许使用NULL值，默认值为NULL；如果带NOT NULL属性，默认值为0。

**存储空间占用量：**2个字节。

**同义词：**INT2 [(M)]。

## MEDIUMINT [(M)]

**含义：**一个中等尺寸的整数。

**可用属性：**AUTO\_INCREMENT、UNSIGNED、ZEROFILL。

**表示范围：**带符号： $-8\,388\,608 \sim 8\,388\,607$  ( $-2^{23} \sim 2^{23}-1$ )；无符号： $0 \sim 16\,777\,215$  ( $0 \sim 2^{24}-1$ )。

**默认值：**如果数据列允许使用NULL值，默认值为NULL；如果带NOT NULL属性，默认值为0。

**存储空间占用量：**3个字节。

**同义词：**INT3 [(M)]和MIDDLEINT [(M)]。

## INT [(M)]

**含义：**一个标准大小的整数。

**可用属性：**AUTO\_INCREMENT、UNSIGNED、ZEROFILL。

**表示范围：**带符号： $-2\,147\,483\,648 \sim 2\,147\,483\,647$  ( $-2^{31} \sim 2^{31}-1$ )；无符号： $0 \sim 4\,294\,967\,295$  ( $0 \sim 2^{32}-1$ )。

**默认值：**如果数据列允许使用NULL值，默认值为NULL；如果带NOT NULL属性，默认值为0。

**存储空间占用量：**4个字节。

**同义词：**INTERGER [(M)]和INT4 [(M)]。

## BIGINT [(M)]

**含义：**一个大整数。

**可用属性：**AUTO\_INCREMENT、UNSIGNED、ZEROFILL。

**表示范围：**带符号： $-9\,223\,372\,036\,854\,775\,808 \sim 9\,223\,372\,036\,854\,775\,807$  ( $-2^{63} \sim 2^{63}-1$ )；无符号： $0 \sim 18\,446\,744\,073\,709\,551\,615$  ( $0 \sim 2^{64}-1$ )。

**默认值：**如果数据列允许使用NULL值，默认值为NULL；如果带NOT NULL属性，默认值为0。

**存储空间占用量：**8个字节。

**同义词：**INT8 [(M)]。

### FLOAT (p)

**含义：**一个浮点数；精确度由 $p$ 指定。如果 $p$ 值在0~24之间，浮点值将被看做是单精度浮点数，即相当于不带 $M$ 和 $D$ 参数的FLOAT类型；如果 $p$ 值在25~53之间，浮点值将被看做是双精度浮点数，即相当于不带 $M$ 和 $D$ 参数的DOUBLE类型。

**可用属性：**UNSIGNED（从MySQL 4.0.2版本开始）、ZEROFILL。

**表示范围：**参见后面对FLOAT和DOUBLE类型的描述。

**默认值：**如果数据列允许使用NULL值，默认值为NULL；如果带NOT NULL属性，默认值为0。

**存储空间占用量：**单精度值需要4个字节，双精度值需要8个字节。

**其他事项：**在MySQL 3.23.6之前的版本里， $p$ 值只能取4或8。从MySQL 3.23.0到3.23.5版本中，FLOAT(4)和FLOAT(8)相当于单精度浮点数和双精度浮点数，而浮点值本身将被保存为硬件所支持的最大精确度。MySQL 3.23.0之前版本里，FLOAT(4)和FLOAT(8)相当于FLOAT(10, 2)和DOUBLE(16, 4)，即浮点值的小数部分将分别被取舍为2位和4位数字，而不是被保存为硬件所支持的最大精确度。

### FLOAT [(M, D)]

**含义：**一个单精度浮点数（精度小于DOUBLE类型）。 $M$ 是显示宽度； $D$ 是小数点后面的精确位数。如果 $D$ 等于0，数据将没有小数点或小数部分。如果 $M$ 和 $D$ 都被省略，显示宽度和小数精度将不确定。

**可用属性：**UNSIGNED（从MySQL 4.0.2版本开始）、ZEROFILL。

**表示范围：**最小非零值： $\pm 1.175\ 494\ 351\text{E}-38$ ；最大非零值： $\pm 3.402\ 823\ 466\text{E}+38$ 。如果给一个浮点数据列加上UNSIGNED属性，它就不允许再容纳负值。

**默认值：**如果数据列允许使用NULL值，默认值为NULL；如果带NOT NULL属性，默认值为0。

**存储空间占用量：**4个字节。

**同义词：**在MySQL 3.23.6之前的版本里，FLOAT和FLOAT4是FLOAT(10, 2)的同义词。

**其他事项：**在MySQL 3.23.6之前的版本里，FLOAT ( $M$ ,  $D$ )数据列里的值将被四舍五入到 $D$ 位小数精度，而不是被保存为硬件所支持的最大精确度。

### DOUBLE [(M, D)]

**含义：**一个双精度浮点数（精度大于FLOAT类型）。 $M$ 是显示宽度； $D$ 是小数点后面的精确位数。如果 $D$ 等于0，数据将没有小数点或小数部分。如果 $M$ 和 $D$ 省略，显示宽度和小数精度将不确定。

**可用属性：**UNSIGNED（从MySQL 4.0.2版本开始）、ZEROFILL。

**表示范围：**最小非零值： $\pm 2.225\ 073\ 858\ 507\ 201\ 4\text{E}-308$ ；最大非零值： $\pm 1.797\ 693\ 134\ 862\ 315\ 7\text{E}+308$ 。如果给一个浮点数据列加上UNSIGNED属性，它就不允许再容纳负值。

**默认值：**如果数据列允许使用NULL值，默认值为NULL；如果带NOT NULL属性，默认值为0。

**存储空间占用量：**8个字节。

**同义词：**DOUBLE PRECISION [(M, D)]和REAL [(M, D)]是DOUBLE [(M, D)]的同义词。在MySQL 3.23.6之前的版本里，DOUBLE和FLOAT8是DOUBLE(16, 4)的同义词。

**其他事项：**在MySQL 3.23.6之前的版本里，DOUBLE(M, D)数据列里的值将被四舍五入到D位小数精度，而不是被保存为硬件所支持的最大精确度。

## DECIMAL [(M, [D] )]

**含义：**一个以字符串形式表示的浮点数（小数点、负号“-”及每个数字都各占用一个字节）。M是显示宽度；D是小数点后面的精确位数。如果D等于0，数据将没有小数点或小数部分。如果M和D省略，显示宽度和小数精度将被默认地设置为10和0。（在MySQL 3.23.6之前的版本里，M和D必须给定，不允许省略。）

**可用属性：**UNSIGNED（从MySQL 4.0.2版本开始）、ZEROFILL。

**表示范围：**最大表示范围与DOUBLE类型相同；有效表示范围由M和D的值决定。

**默认值：**如果数据列允许使用NULL值，默认值为NULL；如果带NOT NULL属性，默认值为0。

**存储空间占用量：**一般说来，一个DECIMAL(M, D)类型的值总共要占用M+2个字节（多出来的两个字节将用来存放小数点和负号“-”）。但是，如果某个DECIMAL数据列是UNSIGNED，就不需要为正负号留出一个字节，从而使存储空间占用量减少一个字节；如果D等于0，就不需要为小数点留出一个字节，这又能使存储空间占用量减少一个字节。在MySQL 3.23之前的版本里，每个DECIMAL(M, D)值将占用M个字节的存储空间，负号和小数点（如果有的话）也包括在这M个字节里。也就是说，在MySQL 3.23及以后的版本里，DECIMAL(M, D)类型的表示范围变大了，存储空间占用量也变大了。

**同义词：**NUMERIC [(M, [D])]和DEC [(M, [D])]

**其他事项：**为与ANSI SQL保持一致，从MySQL 3.23开始，用来存放小数点和正负号的字节不再包括在M值给定的字节里。

## B.2 字符串类型

MySQL字符串类型通常用来存放文本，但它们同时也是能够用来存放任何数据的通用类型。这些类型可以用来存放各种长度的数据值，可以按区分或者不区分字母大小写情况的方式来进行比较，你只需根据具体情况从中挑选出一种来使用就行了。

从MySQL 4.1开始，可以给每一个CHAR、VARCHAR或TEXT类型的数据列分别指定一个专用的字符集。这个操作的语法是CHARACTER SET charset，其中的charset必须是一个合法的字符集标识符，如latin1、greek或utf8等。发出SHOW CHARACTER SET语句能够把服务器当前支持的字符集全都列出来。需要注意的是，如果你给CHAR或VARCHAR数据列指定了一个专用的字符集，就不允许再给这个数据列设定上CHAR或VARCHAR类型通常允许使用的BINARY



属性了。

### CHAR [(M)]

**含义：**一个固定长度的字符串，长度在0~ $M$ 之间。 $M$ 可以是0~255（在MySQL 3.23之前的版本里为1~255）之间的任何整数。长度大于 $M$ 个字节的字符串将在存储时被截断到 $M$ 个字节长。长度小于 $M$ 个字节的字符串将在存储时在右侧以空格补足到 $M$ 个字节长。多余的空格将在检索时被去掉。

**可用属性：**BINARY、CHARACTER SET（从MySQL 4.1版本开始）。

**最大长度：**0到 $M$ 个字节。

**默认值：**如果数据列允许为NULL，则默认值为NULL；如果带NOT NULL属性，默认值为空字符串（"）。

**存储空间占用量：** $M$ 个字节。

**比较方式：**如果没有设定BINARY属性，就不区分大小写。

**同义词：**不带参数的CHAR相当于CHAR(1)。BINARY( $M$ )是CHAR( $M$ ) BINARY的同义词。在MySQL 3.23.5及以后的版本里，NCHAR( $M$ )和NATIONAL CHAR( $M$ )是CHAR( $M$ )的同义词。

### VARCHAR [(M)]

**含义：**一个可变长度的字符串，长度在0~ $M$ 之间。 $M$ 可以是0~255（在MySQL 4.0.2之前的版本里为1~255）之间任何一个整数。长度大于 $M$ 个字节的字符串在存储时将被截断到 $M$ 个字节长。多余的空格将在存储时被去掉。（根据计划，未来的MySQL 4.1系列版本将支持让人们自行选择在何时去掉多余空格的做法。但在写这本书的时候，这一目标仍未实现。）

**可用属性：**BINARY、CHARACTER SET（从MySQL 4.1版本开始）。

**最大长度：**0到 $M$ 个字节。

**默认值：**如果数据列允许为NULL，则默认值为NULL；如果带NOT NULL属性，默认值为空字符串（"）。

**存储空间占用量：**数据本身的长度，再加上1个用来记录这个长度值的字节。

**比较方式：**如果没有设定BINARY属性，就不区分大小写。

**同义词：**CHAR VARYING( $M$ )。在MySQL 3.23.5及以后的版本里，NCHAR VARYING( $M$ )和NATIONAL CHAR VARYING( $M$ )是VARCHAR( $M$ )的同义词。

### TINYBLOB

**含义：**一个小的BLOB值。

**可用属性：**除本附录开头部分介绍的全局属性外，没有其他特殊属性。

**最大长度：**0~255（0~ $2^8-1$ ）个字节。

**默认值：**如果数据列允许为NULL，则默认值为NULL；如果带NOT NULL属性，默认值为空字符串（"）。

**存储空间占用量：**数据本身的长度，再加上1个用来记录这个长度值的字节。

**比较方式：**区分大小写。

## BLOB

含义：一个标准大小的BLOB值。

可用属性：除本附录开头部分介绍的全局属性外，没有其他特殊属性。

最大长度：0~65 535 (0~2<sup>16</sup>-1) 个字节。

默认值：如果数据列允许为NULL，则默认值为NULL；如果带NOT NULL属性，默认值为空字符串("")。

存储空间占用量：数据本身的长度，再加上2个用来记录这个长度值的字节。

比较方式：区分大小写。

## MEDIUMBLOB

含义：一个中等大小的BLOB值。

可用属性：除本附录开头部分介绍的全局属性外，没有其他特殊属性。

最大长度：0~16 777 215 (0~2<sup>24</sup>-1) 个字节。

默认值：如果数据列允许为NULL，则默认值为NULL；如果带NOT NULL属性，默认值为空字符串("")。

存储空间占用量：数据本身的长度，再加上3个用来记录这个长度值的字节。

比较方式：区分大小写。

同义词：LONG VARBINARY。

## LOBLOB

含义：一个大的BLOB值。

可用属性：除本附录开头部分介绍的全局属性外，没有其他特殊属性。

最大长度：0~4 294 967 295 (0~2<sup>32</sup>-1) 个字节。

默认值：如果数据列允许为NULL，则默认值为NULL；如果带NOT NULL属性，默认值为空字符串("")。

存储空间占用量：数据本身的长度，再加上4个用来记录这个长度值的字节。

比较方式：区分大小写。

## TINYTEXT

含义：一个小的TEXT值。

可用属性：CHARACTER SET (从MySQL 4.1版本开始)。

最大长度：0~255 (0~2<sup>8</sup>-1) 个字节。

默认值：如果数据列允许为NULL，则默认值为NULL；如果带NOT NULL属性，默认值为空字符串("")。

存储空间占用量：数据本身的长度，再加上1个用来记录这个长度值的字节。

比较方式：不区分大小写。

## TEXT

含义：一个标准大小的TEXT值。

**可用属性:** CHARACTER SET (从MySQL 4.1版本开始)。

**最大长度:** 0~65 535 (0~2<sup>16</sup>-1) 个字节。

**默认值:** 如果数据列允许为NULL, 则默认值为NULL; 如果带NOT NULL属性, 默认值为空字符串("")。

**存储空间占用量:** 数据本身的长度, 再加上2个用来记录这个长度值的字节。

**比较方式:** 不区分大小写。

## MEDIUMTEXT

**含义:** 一个中等大小的TEXT值。

**可用属性:** CHARACTER SET (从MySQL 4.1版本开始)。

**最大长度:** 0~16 777 215 (0~2<sup>24</sup>-1) 个字节。

**默认值:** 如果数据列允许为NULL, 则默认值为NULL; 如果带NOT NULL属性, 默认值为空字符串("")。

**存储空间占用量:** 数据本身的长度, 再加上3个用来记录这个长度值的字节。

**比较方式:** 不区分大小写。

**同义词:** LONG VARCHAR。

## LONGTEXT

**含义:** 一个大的TEXT值。

**可用属性:** CHARACTER SET (从MySQL 4.1版本开始)。

**最大长度:** 0~4 294 967 295 (0~2<sup>32</sup>-1) 个字节。

**默认值:** 如果数据列允许为NULL, 则默认值为NULL; 如果带NOT NULL属性, 默认值为空字符串("")。

**存储空间占用量:** 数据本身的长度, 再加上4个用来记录这个长度值的字节。

**比较方式:** 不区分大小写。

## ENUM ('value1', 'value2', ...)

**含义:** 一个枚举值; 数据列的取值必须是且仅是合法取值列表中的一个成员。

**可用属性:** 除本附录开头部分介绍的全局属性外, 没有其他特殊属性。

**默认值:** 如果数据列允许为NULL, 默认值为NULL; 如果带NOT NULL属性, 则为枚举集合中的第1个成员。

**存储空间占用量:** 如果成员个数在1~255之间, 占用1个字节; 如果成员个数在256~65 535之间, 占用2个字节。

**比较方式:** 不区分大小写 (MySQL 3.22.1之前的版本区分大小写)。

## SET ('value1', 'value2', ...)

**含义:** 一个集合; 数据列的取值可以是任何一种由该集合0个或者多个成员构成的子集。

**可用属性:** 除本附录开头部分介绍的全局属性外, 没有其他特殊属性。

**默认值:** 如果数据列允许为NULL, 则默认值为NULL; 如果带NOT NULL属性, 默认值为

空集 (")。

**存储空间占用量：**1个字节（1~8个成员的集合）、2个字节（9~16个成员）、3个字节（17~24个成员）、4个字节（25~32个成员）或8个字节（33~64个成员）。

**比较方式：**不区分大小写（MySQL 3.22.1之前的版本区分大小写）。

### B.3 日期和时间类型

利用MySQL提供的与日期和时间有关的各种数据列类型，我们就能把日期和时间值以各种形式表示出来。这些类型包括：DATE（日期）、TIME（时间）、DATETIME（日期+时间）、TIMESTAMP（时间戳）和YEAR（年）。TIMESTAMP类型会在数据记录被修改时自动刷新为当前时刻，而YEAR类型则给那些不需使用完整日期和时间的人们提供了方便。

在以下内容里，日期格式中的CC、YY、MM、DD分别代表世纪、年、月、日；时间格式中的hh、mm、ss分别代表着小时、分钟、秒。

#### DATE

**含义：**一个日期值，格式为'CCYY-MM-DD'。

**可用属性：**除本附录开头部分介绍的全局属性外，没有其他特殊属性。

**表示范围：**'1000-01-01' 到 '9999-12-31'。

**零值：**'0000-00-00'。

**默认值：**如果数据列允许使用NULL值，则为NULL；如果带NOT NULL属性，则为零值。

**存储空间占用量：**3个字节（MySQL 3.23之前的版本需要占用4个字节）。

#### TIME

**含义：**一个时间值，格式为'hh:mm:ss'（负值被表示为'-hh:mm:ss'格式）。它指的是流逝的时间，但可以被当做“一天当中的某一时刻”来看待。

**可用属性：**除本附录开头部分介绍的全局属性外，没有其他特殊属性。

**表示范围：**'-838:59:59' 到 '838:59:59'。

**零值：**'00:00:00'。

**默认值：**如果数据列允许使用NULL值，则为NULL；如果带NOT NULL属性，则为零值。

**存储空间占用量：**3个字节。

**其他事项：**当你试图把非法值插入TIME数据列时，MySQL实际填入的将是零值'00:00:00'。但值得注意的是：这个零值同时也是TIME类型的正常表示范围内的一个合法值。

#### DATETIME

**含义：**一个日期加时间值，格式为'CCYY-MM-DD hh:mm:ss'。

**可用属性：**除本附录开头部分介绍的全局属性外，没有其他特殊属性。

**表示范围：**'1000-01-01 00:00:00' 到 '9999-12-31 23:59:59'。

**零值：**'0000-00-00 00:00:00'。

**默认值：**如果数据列允许使用NULL值，则为NULL；如果带NOT NULL属性，则为零值。

**存储空间占用量：**8个字节。

### **TIMESTAMP [(M)]**

**含义：**一个时间戳（日期+时间）值，格式为 *CCYYMMDDhhmmss*。TIMESTAMP类型的特点是能够把数据行的创建或修改时间记录下来。首先，把NULL值插入数据表中的任何TIMESTAMP数据列的做法都将使MySQL把当时的日期和时间记录到这个数据列里去；其次，只要改变了其他任何数据列里的值，数据表中的第一个TIMESTAMP数据列就将被刷新为当时的日期和时间。不管显示宽度是多少，TIMESTAMP值总是要占用14个字节的存储空间，也总是要以完整的14位数字（世纪、年、月、日、时、分、秒各用两位数字）精度来参加运算。

**可用属性：**除本附录开头部分介绍的全局属性外，没有其他特殊属性。

**表示范围：**19700101000000到2037年的某个时刻。

**零值：**00000000000000。

**默认值：**如果数据列允许使用NULL值，则为NULL；如果带NOT NULL属性，则为零值。

**存储空间占用量：**4个字节。

**其他事项：**如果指定了NOT NULL属性，它将忽略。

### **YEAR [(M)]**

**含义：**一个年份值。如果给出的话，*M*只能是2（以两位数YY表示年份）或者4（以四位数CCYY表示年份）；如果省略，*M*的默认值是4。

**可用属性：**除本附录开头部分介绍的全局属性外，没有其他特殊属性。

**表示范围：**YEAR(4)类型：1901到2155年以及0000年；YEAR(2)类型：1970到2069年，但只显示最后两位数字。

**零值：**YEAR(4)类型的零值是0000，YEAR(2)类型的零值是00。

**默认值：**如果数据列允许使用NULL值，则为NULL；如果带NOT NULL属性，则为零值。

**存储空间占用量：**1个字节。





## 附录C 操作符与函数用法指南

本附录对MySQL提供的操作符和函数做了详细的介绍，SQL语句中的表达式就是通过它们构造出来的。如果没有特殊指明，在本附录里列出的操作符和函数至少从MySQL 3.22.0版本开始就已经存在了。如果大家在实际工作中发现某个操作符或函数的行为与本附录中的描述不一样，请参照*MySQL Reference Manual*（MySQL参考手册）中的有关内容进行核对，看你正在使用的MySQL版本是否对它的行为进行了改动。

本附录中的操作符和函数用法示例将按以下形式写出：

**expression** → **result**

*expression*是一个用来演示有关操作符或函数用法的表达式，*result*是该表达式的求值结果。例如：

`LOWER('ABC')` → 'abc'

上面这行文字的意思是：函数调用`LOWER('ABC')`将产生一个字符串结果'abc'。本附录中的示例都可以在mysql客户程序里试用和检验。具体做法是：启动mysql客户程序，然后依次敲入关键字SELECT、示例表达式和一个分号，最后再按下回车键。比如下面这样：

```
mysql> SELECT LOWER('ABC');
+-----+
| LOWER('ABC') |
+-----+
| abc          |
+-----+
```

MySQL不要求SELECT语句必须有一个FROM子句。这为希望通过输入各种表达式来熟悉操作符和函数使用方法的人们提供了方便。（有的数据库系统不允许使用不带FROM子句的SELECT语句。）

本附录中的某些函数用法示例给出了完整的SELECT语句和它的执行结果。这样的示例在第C.2.6节出现的比较多——如果不这样做，就很难演示出那些统计函数的用法。

函数名和单词形式的操作符（比如BETWEEN）允许以任意的字母大小写形式给出。

本附录还使用了以下几种符号来代表常用的函数参数类型：

- *expr*——代表一个表达式。根据上下文，它可以是数值表达式、字符串表达式或者日期/时间表达式；它还可以是常数、数据列的名字或者其他表达式。
- *str*——代表一个字符串。它可以是一个纯粹的字符串、某字符串值数据列的名字或者一个求值结果为字符串的表达式。
- *n*——代表一个整数（必要时，在字母表里与*n*邻近的字母也将用来代表整数）。
- *x*——代表一个浮点数（必要时，在字母表里与*x*邻近的字母也将用来代表浮点数）。

至于那些不怎么常用的参数，我们将在讨论过程中随时定义。操作符或函数调用中的可选项放在方括号（[]）里。

表达式的求值过程往往会牵涉到有关数据的类型转换问题。本书第2章对引发类型转换动作的上下文环境和MySQL用来把数据值从一种类型转换为另一种类型的有关规则做了详细的讨论。

## C.1 操作符

数据项必须通过操作符才能连接在一起并构成表达式。MySQL为我们准备了很多种操作符，它们可以用来完成算术运算、数据比较、二进制位操作和逻辑操作、模式匹配等。

### C.1.1 操作符的优先级

操作符有着各种各样的优先级。我们把操作符按优先级从高到低的顺序依次列在下面的清单里。在这份清单里，同一行上的操作符都有着相同的优先级。优先级相同的操作符将按从左至右的顺序依次得到求值，而优先级较高的操作符将在优先级较低的操作符之前得到求值。

优 先 级	操 作 符
高	BINARY、COLLATE
	NOT、!
	^
	XOR
	-（一元求负操作符）、~（一元位求反操作符）
	*, /、%
	+, -
	<<、>>
	&
	<、<=、=、<=>、!=、< >、>=、>、IN、IS、LIKE、REGEXP、RLIKE
	BETWEEN、CASE、WHEN、THEN、ELSE
	AND、&&
	OR、
低	:=

与二元操作符相比，一元操作符（-、~、NOT、BINARY）有着更紧密的绑定性。这句话的意思是：在表达式里，一元操作符将先与紧随其后的数据项结合为一个整体，然后再一同参加表达式里的其他操作。如下所示：

-2+3	→ 1
-(2+3)	→ -5

### C.1.2 归组操作符

括号可以用来对表达式的“零件”进行归组。在没有使用括号的表达式里，数据项的求值次序将完全由操作符的优先级（见第C.1.1节）来决定，但括号能改写操作符的优先级并改变表

达式的求值次序。括号还能增加表达式的可读性，使它们更清晰易懂。

$1 + 2 * 3 / 4$	→ 2.50
$((1 + 2) * 3) / 4$	→ 2.25

### C.1.3 算术操作符

这些操作符能够完成标准的算术运算。算术操作符要求操作数必须是数值而不能是字符串，但MySQL会把那些看起来像数值的字符串自动转换为相应的数值。如果操作数中出现NULL值，那么各种算术运算的结果都将是NULL。

#### • +

加法操作符。求值结果为两个操作数的和。

$2 + 2$	→ 4
$3.2 + 4.7$	→ 7.9
$'43bc' + '21d'$	→ 64
$'abc' + 'def'$	→ 0

请注意，最后一个示例表明MySQL不像某些程序设计语言那样把“+”也用做字符串合并操作符。在MySQL里，字符串在参加算术运算之前会先被转换为数值。那些看起来不像是数值的字符串都将被强行转换为0。顺便说一句，如果你真的想合并字符串，就应该使用CONCAT()函数。

#### • -

如果出现在两个表达式之间，就是减法操作符，求值结果为这两个操作数的差；如果出现在单个表达式的前面，就是一元求负操作符，求值结果为操作数的负值（也就是翻转操作数的正负符号）。

$10 - 7$	→ 3
$-(10 - 7)$	→ -3

#### • \*

乘法操作符。求值结果为两个操作数的积。

$2 * 3$	→ 6
$2.3 * -4.5$	→ -10.35

#### • /

除法操作符。求值结果为两个操作数的商。若除数为零，则结果为NULL。

$3 / 1$	→ 3.00
$1 / 3$	→ 0.33
$1 / 0$	→ NULL

#### • %

求余操作符。求值结果为整数 $m$ 除以整数 $n$ 的余数。 $m \% n$ 等同于MOD( $m, n$ )。类似于除法的情况，若除数为0，则结果为NULL。

12 % 4	→ 0
12 % 5	→ 2
12 % 0	→ NULL

如果两个操作数都是整数，它们的加法（+）、减法（-）和乘法（\*）运算就都将在BIGINT值（64位整数）上进行。其中隐藏着一个问题：如果运算结果超出64位整数的表示范围，那么运算结果将不可预料，如下所示：

999999999999999999 * 999999999999999999	→ -7527149226598858751
999999999999 * 999999999999 * 999999999999	→ -1504485813132150785
18014398509481984 * 18014398509481984	→ 0

对于/和%操作符，如果上下文要求运算结果转换为整数，那么除法操作将使用BIGINT值来进行；否则，将使用浮点数来进行。

#### C.1.4 比较操作符

如果比较操作的结果为真，比较操作符的返回值就将是1；如果比较操作的结果为假，比较操作符的返回值就将是0。MySQL允许你对数值或字符串进行比较，并根据以下原则对操作数进行相应的类型转换：

- 除<=>操作符以外，所有涉及NULL值的比较操作都将被求值为NULL。（<=>与=功能相当，但可以用来比较NULL值。表达式NULL <=> NULL将被求值为真。）
- 如果两个操作数都是同类型的字符串值，它们之间的比较操作将按该类型的方式进行。对于二进制字符串，比较操作将一个字节一个字节地比较它们各个字节里的数值；对于非二进制字符串，比较操作将一个字符一个字符地按照它们在有关字符集里的排序次序进行比较。如果两个字符串使用的字符集不同（这种情况可能出现在MySQL 4.1及以后的版本里），比较操作可能不会求值出有意义的结果。如果两个操作数一个是普通的字符串，另一个是二进制字符串，它们之间的比较操作将按二进制字符串方式进行。
- 如果两个操作数都是整数，它们之间的比较操作将按数值方式进行。
- 在MySQL 3.23.22及以后的版本里，比较操作中的十六进制常数将被视为数值。而在此之前，只要十六进制常数不是与数值进行比较，就都视为二进制字符串。
- 如果比较操作中的两个操作数一个是TIMESTAMP或DATETIME值，另一个是常数，比较操作就会把它们都看做是TIMESTAMP值。这是为了使MySQL的比较操作能够与各种ODBC应用程序有更好的配合。
- 如果以上规则都不适用，比较操作中的操作数就都将被视为浮点数。注意，字符串与数值之间的比较操作也落在最后这种情况里。字符串将被转换为一个数值——如果字符串看起来不像是一个数字，转换结果就将是0。比如说，字符串'14.3'将被转换为浮点数14.3，但字符串'L4.3'却会被转换为数值0。

下面的例子可以帮助我们进一步理解上面这些规则：

2 < 12	→ 1
'2' < '12'	→ 0
'2' < 12	→ 1

在第一个比较操作里，两个操作数都是整数，所以它们按数值方式进行比较。在第二个比较操作里，两个操作数都是字符串，所以它们按字符串方式进行比较。在第三个比较操作里，两个操作数一个是整数，另一个是字符串，所以它们被当做浮点数进行比较。

在字符串方式的比较操作里，如果两个操作数都不是二进制字符串，就不区分字母的大小写情况。因此，如果你在字符串比较操作中使用了BINARY关键字，或者参加比较的操作数至少有一个来自于CHAR BINARY、VARCHAR BINARY或BLOB数据列，这个比较操作就将区分字母的大小写情况。

#### • =

如果两个操作数相等，则求值结果为1；否则，求值结果为0。

1 = 1	→ 1
1 = 2	→ 0
'abc' = 'abc'	→ 1
'abc' = 'def'	→ 0
'abc' = 'ABC'	→ 1
BINARY 'abc' = 'ABC'	→ 0
BINARY 'abc' = 'abc'	→ 1
'abc' = 0	→ 1

在默认的情况下，字符串比较操作是不区分字母大小写情况的，所以表达式'abc' = 'abc'和'abc' = 'ABC'的比较结果都为真。但如果加上了BINARY操作符，字符串比较操作就要区分字母的大小写情况了。根据前面介绍的类型转换规则，字符串与数值进行比较的时候将被转换为数值；但因为'abc'看起来根本就不像是一个数值，所以它在比较操作中将被转换为0，这就使'abc' = 0的比较结果也为真。

在比较操作中，形状相似但重音标记不同的字符（比如“E”和“É”）将根据它们在当前字符集中的排序次序来进行比较。

#### • <=>

等于操作符，但允许操作数为NULL值。它的求值情况与=操作符相似，只要两个操作数相等——即使它们是NULL值，求值结果也为1。

1 <=> 1	→ 1
1 <=> 2	→ 0
NULL <=> NULL	→ 1
NULL = NULL	→ NULL

最后两个示例演示了=与<=>在操作数为NULL值时的区别。

<=>操作符最早出现于MySQL 3.23.0版本。

#### • != 或者 <>

如果两个操作数不相等，则求值结果为1；否则，求值结果为0。

3.4 != 3.4	→ 0
'abc' <> 'ABC'	→ 0
BINARY 'abc' <> 'ABC'	→ 1
'abc' != 'def'	→ 1



## • &lt;

如果左操作数小于右操作数，则求值结果为1；否则，求值结果为0。

3 < 10	→ 1
105.4 < 10e+1	→ 0
'abc' < 'ABC'	→ 0
'abc' < 'def'	→ 1

## • &lt;=

如果左操作数小于或等于右操作数，则求值结果为1；否则，求值结果为0。

'abc' <= 'a'	→ 0
'a' <= 'abc'	→ 1
13.5 <= 14	→ 1
(3 * 4) - (6 * 2) <= 0	→ 1

## • &gt;=

如果左操作数大于或等于右操作数，则求值结果为1；否则，求值结果为0。

'abc' >= 'a'	→ 1
'a' >= 'abc'	→ 0
13.5 >= 14	→ 0
(3 * 4) - (6 * 2) >= 0	→ 1

## • &gt;

如果左操作数大于右操作数，则求值结果为1；否则，求值结果为0。

PI() > 3	→ 1
'abc' > 'a'	→ 1
SIN(0) > COS(0)	→ 0

• *expr* BETWEEN *min* AND *max*

*expr* NOT BETWEEN *min* AND *max*

如果*expr*落在从*min*到*max*的区间内（包括*min*和*max*在内），则BETWEEN操作符的求值结果为1；否则，求值结果为0。NOT BETWEEN操作符的求值情况正好与此相反。如果操作数*expr*、*min*和*max*都是同一种类型，则下面两个表达式等价：

```
expr BETWEEN min AND max
(min <= expr AND expr <= max)
```

如果这些操作数不是同一种类型，因为会发生类型转换，上面这两个表达式就不一定等价了。此时，BETWEEN操作符的求值结果将由*expr*的类型所决定的比较操作来确定：

- 如果*expr*是一个字符串，这些操作数就将被视为字符串并按字符串方式进行比较。而比较操作是否需要区分字母的大小写情况又进一步取决于*expr*是否是二进制字符串。
- 如果*expr*是一个整数，这些操作数就将被视为整数并按数值方式进行比较。
- 如果以上两条规则都不适用，这些操作数就将被视为浮点数并按数值方式进行比较。

'def' BETWEEN 'abc' and 'ghi'	→ 1
'def' BETWEEN 'abc' and 'def'	→ 1

13.3 BETWEEN 10 and 20	→ 1
13.3 BETWEEN 10 and 13	→ 0
2 BETWEEN 2 and 2	→ 1
'B' BETWEEN 'A' and 'a'	→ 0
BINARY 'B' BETWEEN 'A' and 'a'	→ 1

• CASE *expr* WHEN *expr1* THEN *result1* ...

[ ELSE *default* ] END

CASE WHEN *expr1* THEN *result1* ...

[ ELSE *default* ] END

我们先来看CASE操作符的第一种形式。它将把表达式*expr*与每一个WHEN后面的表达式(*expr1*、*expr2*……)依次进行比较,当两者相等时,相应的THEN后面的值就是CASE操作符的求值结果。这特别适用于需要把一个给定值与一组值进行比较的场合:

CASE 0 WHEN 1 THEN 'T' WHEN 0 THEN 'F' END	→ 'F'
CASE 'F' WHEN 'T' THEN 1 WHEN 'F' THEN 0 END	→ 0

我们再来看CASE操作符的第二种形式。它将依次对每一个WHEN后面的表达式(*expr1*、*expr2*……)进行求值,如果结果为真(既不能是0,也不能是NULL),相应的THEN后面的值就是CASE操作符的求值结果。这特别适用于需要判断“不等于”关系或者需要对一系列条件进行测试的场合:

CASE WHEN 1=0 THEN 'absurd' WHEN 1=1 THEN 'obvious' END	→ 'obvious'
---	-------------

如果WHEN后面的表达式没有一个成立,CASE操作符的求值结果就将是ELSE后面的值。如果ELSE子句不存在,CASE操作符的求值结果就将是NULL。

CASE 0 WHEN 1 THEN 'true' ELSE 'false' END	→ 'false'
CASE 0 WHEN 1 THEN 'true' END	→ NULL
CASE WHEN 1=0 THEN 'true' ELSE 'false' END	→ 'false'
CASE WHEN 1/0 THEN 'true' END	→ NULL

第一个THEN后面的值的类型决定着整个CASE表达式的类型。

CASE 1 WHEN 0 THEN 0 ELSE 1 END	→ 1
CASE 1 WHEN 0 THEN '0' ELSE 1 END	→ '1'

CASE操作符最早出现于MySQL 3.23.3。

• *expr* IN (*value1*, *value2*, ...)

*expr* NOT IN (*value1*, *value2*, ...)

如果*expr*与列表中的某个值相等,IN()的求值结果就将为1;否则,就将为0。NOT IN()的求值情况正好与此相反。下面两个表达式是等价的:

<i>expr</i> NOT IN ( <i>value1</i> , <i>value2</i> , ...)
NOT ( <i>expr</i> IN ( <i>value1</i> , <i>value2</i> , ...))

如果列表里的值全都是常数,MySQL就会对它们进行排序并利用二元搜索树算法来对IN()测试进行求值,这个算法是非常快的。

3 IN (1,2,3,4,5)	→ 1
'd' IN ('a','b','c','d','e')	→ 1
'f' IN ('a','b','c','d','e')	→ 0
3 NOT IN (1,2,3,4,5)	→ 0
'd' NOT IN ('a','b','c','d','e')	→ 0
'f' NOT IN ('a','b','c','d','e')	→ 1

#### • *expr* IS NULL

*expr* IS NOT NULL

如果*expr*的值是NULL, IS NULL的求值结果就将为1; 否则, 就将为0。IS NOT NULL的求值情况正好与此相反。下面两个表达式是等价的:

```
expr IS NOT NULL
NOT (expr IS NULL)
```

IS NULL和IS NOT NULL是专门用来测试NULL值的比较操作符。普通的比较操作符=和!=无法进行这种判断。(从MySQL 3.23开始, 还可以用操作符<=>来测试NULL值。)

NULL IS NULL	→ 1
0 IS NULL	→ 0
NULL IS NOT NULL	→ 0
0 IS NOT NULL	→ 1
NOT (0 IS NULL)	→ 1
NOT (NULL IS NULL)	→ 0
NOT NULL IS NULL	→ 1

因为操作符NOT的优先级要高于操作符IS (请参阅第C.1.1节), 所以最后一个例子的求值结果是1。

### C.1.5 位操作符

本小节介绍用来完成各种位操作的操作符。位操作必须在BIGINT值(64位整数)上进行, 这就限制了这类操作的最大范围。如果操作数里有NULL值, 则位操作的结果就将是NULL。

#### • &

求值结果为两个操作数逐位进行AND (与) 操作得到的结果。

1 & 1	→ 1
1 & 2	→ 0
7 & 5	→ 5

#### • |

求值结果为两个操作数逐位进行OR (或) 操作得到的结果。

1   1	→ 1
1   2	→ 3
1   2   4   8	→ 15
1   2   4   8   15	→ 15

#### • ^

求值结果为两个操作数逐位进行XOR (异或) 操作得到的结果。

<code>1 ^ 1</code>	$\rightarrow 0$
<code>1 ^ 0</code>	$\rightarrow 1$
<code>255 ^ 127</code>	$\rightarrow 128$

这个操作符最早出现于MySQL 4.0.2版本。在此之前，可以利用下面这个表达式对值 $m$ 和 $n$ 求出同样的结果来：

`(m & (~n)) | ((~m) & n)`

#### • <<

把左操作数的各个位逐位左移，移动数由右操作数指定。如果右操作数为负值，则操作结果为0。

<code>1 &lt;&lt; 2</code>	$\rightarrow 4$
<code>2 &lt;&lt; 2</code>	$\rightarrow 8$
<code>1 &lt;&lt; 62</code>	$\rightarrow 4611686018427387904$
<code>1 &lt;&lt; 63</code>	$\rightarrow -9223372036854775808$
<code>1 &lt;&lt; 64</code>	$\rightarrow 0$

最后两个例子演示了这个操作符在64位上的极限情况。

<<操作符最早出现于MySQL 3.22.2版本。

#### • >>

把左操作数的各个位逐位右移，移动数由右操作数指定。如果右操作数为负值，则操作结果为0。

<code>16 &gt;&gt; 3</code>	$\rightarrow 2$
<code>16 &gt;&gt; 4</code>	$\rightarrow 1$
<code>16 &gt;&gt; 5</code>	$\rightarrow 0$

>>操作符最早出现于MySQL 3.22.2版本。

#### • ~

对随后的表达式逐位求反，即把所有的0位翻转为1，把所有的1位翻转为0。

<code>~0</code>	$\rightarrow -1$
<code>~(-1)</code>	$\rightarrow 0$
<code>~~(-1)</code>	$\rightarrow -1$

~操作符最早出现于MySQL 3.23.5版本。

### C.1.6 逻辑操作符

逻辑操作符也叫做布尔操作符，它们用来测试某个表达式是否成立（成立为真，不成立为假）。在MySQL里，如果逻辑操作符的求值结果为真，则返回1；如果为假，则返回0。逻辑操作符把非零操作数解释为真，把0操作数解释为假。逻辑操作符对NULL值的处理情况见它们各自的条目。

逻辑操作符要求操作数是数值类型，字符串操作数在求值过程开始之前会被转换为数值。

#### • NOT 或 !

逻辑非操作符。如果随后的操作数为假，则求值结果为1；如果操作数为真，则求值结果

为0。但NOT NULL仍将为NULL。

NOT 0	→ 1
NOT 1	→ 0
NOT NULL	→ NULL
NOT 3	→ 0
NOT NOT 1	→ 1
NOT '1'	→ 0
NOT '0'	→ 1
NOT ''	→ 1
NOT 'abc'	→ 1

#### • AND 或 &&

逻辑与操作符。如果两个操作数都是真（既不能是0，也不能是NULL），则求值结果为1；否则，求值结果为0。

0 AND 0	→ 0
0 AND 3	→ 0
4 AND 2	→ 1

不同MySQL版本里的AND操作符对NULL值（即操作数是一个未知值）的处理方式是不一样的。在MySQL 3.23.9及以后的版本里，如果结果肯定为假，AND操作符的求值结果就将是0；如果结果无法确定，其求值结果就将是NULL。如下所示：

1 AND NULL	→ NULL
0 AND NULL	→ 0
NULL AND NULL	→ NULL

在MySQL 3.23.9之前的版本里，AND操作符会把NULL求值为0。实际上，NULL值是被当做0来对待的。如下所示：

1 AND NULL	→ 0
0 AND NULL	→ 0
NULL AND NULL	→ 0

#### • OR 或 ||

逻辑或操作符。只要两个操作数里有一个为真（既不能是0，也不能是NULL），则求值结果为1；否则，求值结果为0。

0 OR 0	→ 0
0 OR 3	→ 1
4 OR 2	→ 1

不同MySQL版本里的OR操作符对NULL值（即操作数是一个未知值）的处理方式是不一样的。在MySQL 3.23.9及以后的版本里，如果结果肯定为真，OR操作符的求值结果就将是1；如果结果无法确定，其求值结果就将是NULL。如下所示：

1 OR NULL	→ 1
0 OR NULL	→ NULL
NULL OR NULL	→ NULL



在MySQL 3.23.9之前的版本里，如果结果肯定为真，OR操作符的求值结果就将是1；否则，其求值结果就将是0。实际上，NULL值是被当做0来对待的。如下所示：

1 OR NULL	→ 1
0 OR NULL	→ 0
NULL OR NULL	→ 0

#### • XOR

逻辑异或操作符。如果有且仅有一个操作数为真（既不能是0，也不能是NULL），则求值结果为1；否则，求值结果为0。如果操作数中有NULL值，求值结果为NULL。

0 XOR 0	→ 0
0 XOR 9	→ 1
7 XOR 0	→ 1
5 XOR 2	→ 0

XOR操作符最早出现于MySQL 4.0.2版本。在此之前，可以利用下面这个表达式对值 $m$ 和 $n$ 求出同样的结果来：

$(m \text{ AND } (\text{NOT } n)) \text{ OR } ((\text{NOT } m) \text{ AND } n)$

MySQL允许使用C语言中的逻辑操作符！（逻辑非）、||（逻辑或）和&&（逻辑与）。需要提醒大家注意的是，MySQL不像某些SQL版本那样把||用做字符串合并操作符。如果你真的想合并字符串，应该使用CONCAT()函数。（如果你真的想把||当做字符串合并操作符来使用，可以在启动MySQL服务器的时候使用--ansi选项来激活ANSI模式。）

### C.1.7 类型转换操作符

类型转换操作符能够把数据值从一个类型转换为另一个类型。

#### • `_charset str`

这个操作符用来把一个字符串常数或者某个数据列里的值转换到一个给定的字符集上去，`charset`必须是服务器所支持的某个字符集的名字。比如说，下面三个表达式将分别把字符串'abc'转换到latin1\_de、utf8或ucs2字符集上：

```
_latin1_de 'abc'
_utf8 'abc'
_ucs2 'abc'
```

`_charset`操作符最早出现于MySQL 4.1.0版本。

#### • `BINARY str`

BINARY操作符用来把紧随其后的操作数转换为一个二进制字符串，其效果是使该字符串上的比较操作区分字母的大小写情况。如果紧随其后的操作数是一个数值，就先把它转换为字符串形式：

'abc' = 'ABC'	→ 1
'abc' = BINARY 'ABC'	→ 0
BINARY 'abc' = 'ABC'	→ 0
'2' < 12	→ 1
'2' < BINARY 12	→ 0

在最后一个例子里，BINARY操作符强制进行一次由数值到字符串的转换。然后，因为两个操作数都是字符串，所以比较操作将按字符串方式进行。

BINARY操作符最早出现于MySQL 3.23.0版本。

• *str COLLATE charset*

COLLATE操作符将使给定字符串*str*按字符集*charset*的排序次序进行比较。这对比较操作、排序操作、归组操作以及DISTINCT等操作都会产生影响，如下所示：

```
SELECT ... WHERE col_name COLLATE utf8 > 'M';
SELECT MAX(col_name COLLATE greek) FROM ... ;
SELECT ... GROUP BY col_name COLLATE latin1;
SELECT ... ORDER BY col_name COLLATE czech;
SELECT DISTINCT col_name COLLATE latin1_de FROM ...;
```

COLLATE操作符最早出现于MySQL 4.1.0版本。

### C.1.8 模式匹配操作符

MySQL提供了两种模式匹配机制：一种是使用LIKE操作符的SQL模式匹配，另一种是使用REGEXP操作符的正则表达式模式匹配。这两种模式匹配机制的主要区别是：1) LIKE操作符一般不区分字母的大小写情况，除非它的操作数里至少有一个是二进制字符串。REGEXP操作符也是如此，但在MySQL 3.23.4之前的版本里，REGEXP操作符却一直是区分字母的大小写情况的；2) 只有整个字符串得到匹配时，LIKE操作符才算匹配成功。而只要能在字符串里找到匹配模式，REGEXP操作符就算匹配成功。

• *str LIKE pat [ESCAPE 'c']*

*str NOT LIKE pat [ESCAPE 'c']*

LIKE是SQL模式匹配操作的操作符，当匹配模式*pat*与字符串表达式*str*完全匹配时，它的求值结果将是1。如果模式没有得到匹配，LIKE操作符的求值结果将是0。NOT LIKE操作符的求值情况则正好相反。下面两个表达式是等价的：

```
str NOT LIKE pat [ESCAPE 'c']
NOT (str LIKE pat [ESCAPE 'c'])
```

只要两个操作数里有一个是NULL，求值结果就将是NULL。

在SQL模式里，有两个字符是有着特殊含义的通配符：

- “%”——能与除NULL以外的任意长度的字符序列（包括空字符序列）相匹配。
- “\_”——能与任何一个单个的字符相匹配。

SQL模式允许混合使用这两种通配符，如下所示：

'catnip' LIKE 'cat%'	→ 1
'dogwood' LIKE '%wood'	→ 1
'bird' LIKE '____'	→ 1
'bird' LIKE '____'	→ 0
'dogwood' LIKE '%wo__'	→ 1

用来进行SQL模式匹配的LIKE操作符是否区分字母的大小写情况要取决于它的操作数。

在一般情况下，LIKE操作符不区分字母的大小写情况。但只要它的两个操作数里有二进制字符串，LIKE操作符就会区分字母的大小写情况：

```
'abc' LIKE 'ABC' → 1
BINARY 'abc' LIKE 'ABC' → 0
'abc' LIKE BINARY 'ABC' → 0
```

因为通配符“%”能够与任何一个字符序列相匹配，所以它甚至能与空字符串相匹配：

```
' ' LIKE '%' → 1
'cat' LIKE 'cat%' → 1
```

MySQL允许使用LIKE操作符对数值表达式进行匹配：

```
50 + 50 LIKE '1%' → 1
200 LIKE '2__' → 1
```

如果想对通配符“%”或“\_”本身进行匹配，就必须给它们加上一个前导的反斜线字符（即写成“\%”或“\\_”的样子）以取消其特殊含义，如下所示：

```
'100% pure' LIKE '100%' → 1
'100% pure' LIKE '100\%' → 0
'100% pure' LIKE '100\% pure' → 1
```

如果你不想使用默认的转义字符“\”，可以在ESCAPE子句里另行指定一个，如下所示：

```
'100% pure' LIKE '100^%' ESCAPE '^' → 0
'100% pure' LIKE '100^% pure' ESCAPE '^' → 1
```

#### • *str* REGEXP *pat*

*str* NOT REGEXP *pat*

REGEXP是正则表达式模式匹配操作的操作符，只要能在字符串表达式*str*里找到匹配模式*pat*，它的求值结果将是1；否则，它的求值结果就将是0。NOT REGEXP操作符的求值情况正好与REGEXP操作符相反。下面两个表达式是等价的：

```
str NOT REGEXP pat
NOT (str REGEXP pat)
```

REGEXP操作符使用的正则表达式与grep和sed等UNIX程序所使用的匹配模式非常相似。表C-1列出了REGEXP操作符的各种匹配序列。

表C-1 正则表达式元素

元 素	含 义
^	匹配字符串的开始
\$	匹配字符串的结尾
.	匹配任何单个的字符，包括换行符
[...]	匹配方括号内的任何一个字符
[^...]	匹配没有出现在方括号内的任何一个字符
e*	匹配模式元素e的0个或多个实例
e+	匹配模式元素e的1个或多个实例

(续)

元 素	含 义
$e?$	匹配模式元素 $e$ 的0个或1个实例
$e1 e2$	匹配模式元素 $e1$ 或 $e2$
$e\{m\}$	匹配模式元素 $e$ 的 $m$ 个实例
$e\{m,\}$	匹配模式元素 $e$ 的 $m$ 个或多个实例
$e\{,n\}$	匹配模式元素 $e$ 的0个到 $n$ 个实例
$e\{m,n\}$	匹配模式元素 $e$ 的 $m$ 个到 $n$ 个实例
$(\dots)$	把括号中的模式元素当做一个模式元素来对待
其他	非特殊字符将匹配其自身

只要两个操作数里有一个是NULL, REGEXP操作符的求值结果就将是NULL。

REGEXP操作符不要求匹配模式与整个字符串相匹配, 只要能在字符串里找到匹配模式就足够了, 如下所示:

```
'cats and dogs' REGEXP 'dogs'      → 1
'cats and dogs' REGEXP 'cats'      → 1
'cats and dogs' REGEXP 'c.*a.*d'   → 1
'cats and dogs' REGEXP 'o'         → 1
'cats and dogs' REGEXP 'x'         → 0
```

' $^pat$ '和' $pat\$$ '形式的模式分别用来匹配' $pat$ '模式出现在字符串的开头或末尾的情况; 而' $^pat\$$ '形式的模式可以用来匹配' $pat$ '模式完全匹配整个字符串的情况, 如下所示:

```
'abcde' REGEXP 'b'                  → 1
'abcde' REGEXP '^b'                 → 0
'abcde' REGEXP 'b\$'                → 0
'abcde' REGEXP '^a'                 → 1
'abcde' REGEXP 'e\$'                → 1
'abcde' REGEXP '^a.*e\$'             → 1
```

" $[...]$ " 或 " $^ [...]$ " 构造用来设定字符分组。在字符分组里, 可以用连字符(-)来设定一个字符区间, 连字符的两端是字符区间的起始字符和结尾字符。比如说,  $[a-z]$ 将匹配任何一个小写字母, 而 $[0-9]$ 将匹配任何一个数字:

```
'bin' REGEXP '^b[aeiou]n\$'         → 1
'bxn' REGEXP '^b[aeiou]n\$'         → 0
'oboeist' REGEXP '^ob[aeiou]+st\$'  → 1
'wolf359' REGEXP '[a-z]+[0-9]+'     → 1
'wolf359' REGEXP '[0-9a-z]+'        → 1
'wolf359' REGEXP '[0-9]+[a-z]+'     → 0
```

如果你想把字符 "]" 放到一个字符分组里, 就必须把它放在字符分组的第一个。如果你想把字符 "-" 放到一个字符分组里, 就必须把它放在字符分组的第一个或者最后一个。如果你想把字符 "^" 放到一个字符分组里, 它不能是 "[" 后面的第一个字符。

MySQL允许我们在REGEXP操作符的正则表达式里直接使用几种标准的POSIX专用字符分组, 这些字符分组对有关字符的排序次序做出了安排。一些常用的POSIX字符分组见表C-2。

表C-2 正则表达式POSIX字符分组

元 素	含 义
<code>[:alnum:]</code>	字母和数字字符
<code>[:alpha:]</code>	字母字符
<code>[:blank:]</code>	空白字符（空格或制表符）
<code>[:cntrl:]</code>	控制字符
<code>[:digit:]</code>	十进制数字（0~9）
<code>[:graph:]</code>	图形字符（不包括空白字符）
<code>[:lower:]</code>	小写字母字符
<code>[:print:]</code>	图形或空白字符
<code>[:punct:]</code>	标点符号字符
<code>[:space:]</code>	空格、制表符、换行符或回车符
<code>[:upper:]</code>	大写字母字符
<code>[:xdigit:]</code>	十六进制数字（0~9, a~f, A~F）

在使用时，POSIX专用字符分组要放在REGEXP操作符的字符分组里：

```
'abc' REGEXP '[:space:]' → 0
'a c' REGEXP '[:space:]' → 1
'abc' REGEXP '[:digit][:punct:]' → 0
'a0c' REGEXP '[:digit][:punct:]' → 1
'a,c' REGEXP '[:digit][:punct:]' → 1
```

MySQL还允许我们在REGEXP操作符的正则表达式字符串里使用类似C语言的语法。比如说，`\n`、`\t`、`\\`将分别被解释为换行符、制表符和反斜线字符（`\`）。当需要在匹配模式里使用这几个字符时，必须双写反斜线字符（即把它们分别写成`\\n`、`\\t`、`\\\\`的样子）——语法解释器在对查询语句进行分析时会去掉一个反斜线字符，模式匹配操作再把剩下的转义序列（即`\n`、`\t`、`\\`）解释为相应的字符。

#### • `str RLIKE pat`

`str NOT RLIKE pat`

RLIKE和NOT RLIKE是REGEXP和NOT REGEXP操作符的同义词。

## C.2 函数

函数在调用后会返回一个值。在默认的情况下，函数名与紧随其后的左括号之间不允许出现空格，如下所示：

```
NOW()          /* 正确
NOW ( )        /* 不正确
```

如果你在启动MySQL服务器的时候使用了`--ansi`选项，函数名与紧随其后的左括号之间就允许有空格，但这种做法的副作用是所有的函数名都将被视为保留字。也可以在建立服务器连接的时候激活这种行为：1）在启动mysql客户程序时，给它加上`--ignore-space`选项；2）在C程序里，调用`mysql_real_connect()`函数时给它加上`CLIENT_IGNORE_SPACE`选项。

在大多数场合，可以用逗号来分隔某个函数的多个输入参数，函数参数的前后也允许出现



空格,如下所示:

```
CONCAT('abc','def')      /* 允许这样做
CONCAT('abc','def')      /* 也允许这样做
```

但有些函数不允许这样做,比如TRIM()或EXTRACT()函数:

```
TRIM(' ' FROM 'x ')      → 'x'
EXTRACT(YEAR FROM '2003-01-01') → 2003
```

这类情况将在我们具体介绍到有关函数时加以注明。

### C.2.1 比较类函数

#### • GREATEST(expr1, expr2, ...)

返回值是输入参数中的最大值。这个“最大值”是根据以下原则确定的:

- 如果这个函数是在整数上下文里被调用的或者它的输入参数全都是整数,输入参数就将按整数方式进行比较。
- 如果这个函数是在浮点数上下文里被调用的或者它的输入参数全都是浮点数,输入参数就将按浮点数方式进行比较。
- 如果以上两条规则都不适用,输入参数就将按字符串方式进行比较。只要输入参数里没有二进制字符串,比较操作就不区分字母的大小写情况。如下所示:

```
GREATEST(2,3,1)          → 3
GREATEST(38.5,94.2,-1)   → 94.2
GREATEST('a','ab','abc') → 'abc'
GREATEST(1,3,5)          → 5
GREATEST('A','b','C')    → 'C'
GREATEST(BINARY 'A','b','C') → 'b'
```

GREATEST()函数最早出现于MySQL 3.22.5。在此前的版本里,可以用MAX()函数来代替之。

#### • IF(expr1, expr2, expr3)

若表达式expr1为真(既不能是0,也不能是NULL),则返回expr2;否则,返回expr3。IF()函数将根据自己被调用时的上下文来决定是返回一个数值还是返回一个字符串。如下所示:

```
IF(1,'true','false')    → 'true'
IF(0,'true','false')    → 'false'
IF(NULL,'true','false') → 'false'
IF(1.3,'non-zero','zero') → 'non-zero'
IF(0.3,'non-zero','zero') → 'zero'
IF(0.3 != 0,'non-zero','zero') → 'non-zero'
```

expr1将被求值为一个整数,所以请大家特别注意后三个例子。请看,1.3被转换为整数1,相当于逻辑真值;但0.3却被转换为0,相当于逻辑假值。最后一个例子演示了浮点数在比较操作中的正确用法:用一个比较表达式来测试那些浮点数。只有这样,浮点值才能正确

地按浮点方式进行比较并正确地得出一个以整数值1或0来表示的逻辑真假值——正如IF()函数要求的那样。

- IFNULL(*expr1*, *expr2*)

若表达式*expr1*的值为NULL, 则返回*expr2*; 否则, 返回*expr1*。IFNULL()函数将根据自己被调用时的上下文来决定是返回一个数值还是返回一个字符串。如下所示:

```
IFNULL(NULL, 'null')           → 'null'
IFNULL('not null', 'null')     → 'not null'
```

- INTERVAL(*n*, *n1*, *n2*, ...)

若*n* < *n1*, 则返回0; 若*n* < *n2*, 则返回1; 依此类推。若*n*是NULL值, 则返回-1。*n1*, *n2*, ...必须是一个严格按递增顺序排列的序列——因为这个操作符的比较操作是使用快速的二元搜索算法完成的, 如果不是这样, INTERVAL()函数的行为将难以预料。

```
INTERVAL(1.1, 0, 1, 2)         → 2
INTERVAL(7, 1, 3, 5, 7, 9)     → 4
```

- ISNULL(*expr*)

若表达式*expr*的值是NULL, 则返回1; 否则, 返回0。

```
ISNULL(NULL)                   → 1
ISNULL(0)                       → 0
ISNULL(1)                       → 0
```

- LEAST(*expr1*, *expr2*, ...)

返回值是输入参数中的最小值。用来确定“最小值”的原则与GREATEST()函数的一样。

```
LEAST(2, 3, 1)                 → 1
LEAST(38.5, 94.2, -1)          → -1.0
LEAST('a', 'ab', 'abc')        → 'a'
```

LEAST()函数最早出现于MySQL 3.22.5。在此前的版本里, 可以用MIN()函数来代替之。

- NULLIF(*expr1*, *expr2*)

若作为输入参数的两个表达式的值不同, 返回*expr1*; 若它们的值相同, 则返回NULL。

```
NULLIF(3, 4)                   → 3
NULLIF(3, 3)                   → NULL
```

NULLIF()函数最早出现于MySQL 3.23.19。

- STRCMP(*str1*, *str2*)

这个函数将按字符方式对两个参数进行比较, 并根据字符串*str1*是否小于、等于、大于字符串*str2*而返回1、0、-1。只要两个参数中有一个是NULL值, 这个函数就会返回NULL。在MySQL 4.0.0及以后的版本里, 除非两个参数中至少有一个是二进制字符串, 否则它们的比较操作将不区分字母的大小写情况。

```
STRCMP('a', 'a')               → 0
STRCMP('a', 'A')               → 0
STRCMP('A', 'a')               → 0
STRCMP(BINARY 'a', 'A')       → 1
STRCMP(BINARY 'A', 'a')       → -1
```

在MySQL 4.0.0之前的版本里, 这个函数的比较操作是区分字母大小写情况的。

```
STRCMP('a','a')          → 0
STRCMP('a','A')          → 1
STRCMP('A','a')          → -1
```

### C.2.2 类型转换类函数

这类函数能够把数据值从一种类型转换为另一种类型。

#### • CAST(*expr* AS *type*)

把表达式*expr*的值转换为给定的类型。其中, *type*可以是BINARY(二进制字符串)、DATE、DATETIME、TIME、SIGNED、SIGNED INTEGER、UNSIGNED或UNSIGNED INTEGER。

```
CAST(304 AS BINARY)      → '304'
CAST(-1 AS UNSIGNED)     → 18446744073709551615
```

在需要使用CREATE TABLE... SELECT语句来创建新数据表的时候, 人们经常会利用CAST()函数把某些数据列强行设置为指定的类型。如下所示:

```
mysql> CREATE TABLE t SELECT CAST(20020101 AS DATE) AS date_val;
mysql> SHOW COLUMNS FROM t;
+-----+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| date_val | date |      |      | 0000-00-00 |      |
+-----+-----+-----+-----+-----+-----+
mysql> SELECT * FROM t;
+-----+
| date_val |
+-----+
| 2002-01-01 |
+-----+
```

CAST()函数最早出现于MySQL 4.0.2。CAST()与CONVERT()函数相似, 但前者遵守的是ANSI SQL语法, 而后者遵守的却是ODBC语法。

#### • CONVERT(*expr*, *type*)

CONVERT(*expr* USING *charset*)

除使用的语法稍微有些不同外, CONVERT()函数的第一种形式与CAST()函数的功能完全相同, *expr*和*type*参数的用法也完全一致。第二种形式(USING)用来把表达式*expr*的值转换到指定的字符集上。

```
CONVERT(304, BINARY)      → '304'
CONVERT(-1, UNSIGNED)     → 18446744073709551615
CONVERT('abc' USING utf8); → 'abc'
```

CONVERT()函数最早出现于MySQL 4.0.2, 而使用USING语法的形式最早出现于MySQL 4.1.0。

### C.2.3 数值类函数

数值类函数在出错时将返回NULL。比如说，如果你传递给某个数值类函数的输入参数超出了它的允许范围或者是一个非法值，这个函数就会返回NULL。

- **ABS(x)**

返回值： $x$ 的绝对值。

ABS(13.5)	→ 13.5
ABS(-13.5)	→ 13.5

- **ACOS(x)**

返回值： $x$ 的反余弦值。如果 $x$ 不在-1~1的区间内，则返回NULL。

ACOS(1)	→ 0.000000
ACOS(0)	→ 1.570796
ACOS(-1)	→ 3.141593

- **ASIN(x)**

返回值： $x$ 的反正弦值。如果 $x$ 不在-1~1的区间内，则返回NULL。

ASIN(1)	→ 1.570796
ASIN(0)	→ 0.000000
ASIN(-1)	→ -1.570796

- **ATAN(x)**

ATAN( $y, x$ )

返回值：只带一个输入参数的ATAN( $x$ )函数将返回 $x$ 的反正切值；带两个输入参数的ATAN( $y, x$ )函数是ATAN2函数的同义词。

ATAN(1)	→ 0.785398
ATAN(0)	→ 0.000000
ATAN(-1)	→ -0.785398

- **ATAN2(y, x)**

返回值：相当于ATAN( $y/x$ )，但它要根据两个输入参数的正负符号来判断自己的返回值将落在哪一个坐标象限里。

ATAN2(1, 1)	→ 0.785398
ATAN2(1, -1)	→ 2.356194
ATAN2(-1, 1)	→ -0.785398
ATAN2(-1, -1)	→ -2.356194

- **CEILING(x)**

返回值：不小于 $x$ 的最小整数。这个返回值永远是一个BIGINT类型的值。

CEILING(3.8)	→ 4
CEILING(-3.8)	→ -3

- **COS(x)**

返回值： $x$ 的余弦值； $x$ 被视为是一个弧度值。

<code>COS(0)</code>	→ 1.000000
<code>COS(PI())</code>	→ -1.000000
<code>COS(PI()/2)</code>	→ 0.000000

• **COT(x)**

返回值： $x$ 的余切值； $x$ 被视为是一个弧度值。

<code>COT(PI()/2)</code>	→ 0.00000000
<code>COT(PI()/4)</code>	→ 1.00000000

• **CRC32(str)**

返回值：字符串 $str$ 的循环冗余校验值，这个返回值是一个32位（即 $0 \sim 2^{32}-1$ 之间）的无符号整数值。如果输入参数是NULL值，则返回NULL。

<code>CRC32('xyz')</code>	→ 3951999591
<code>CRC32('0')</code>	→ 4108050209
<code>CRC32(0)</code>	→ 4108050209
<code>CRC32(NULL)</code>	→ NULL

CRC32()函数最早出现于MySQL 4.1.0。

• **DEGREES(x)**

返回值：弧度值 $x$ 的角度值。

<code>DEGREES(PI())</code>	→ 180
<code>DEGREES(PI()*2)</code>	→ 360
<code>DEGREES(PI()/2)</code>	→ 90
<code>DEGREES(-PI())</code>	→ -180

• **EXP(x)**

返回值： $e^x$ （ $e$ 的 $x$ 次方）； $e$ 是自然对数底。

<code>EXP(1)</code>	→ 2.718282
<code>EXP(2)</code>	→ 7.389056
<code>EXP(-1)</code>	→ 0.367879
<code>1/EXP(1)</code>	→ 0.36787944

• **FLOOR(x)**

返回值：不大于 $x$ 的最大整数。这个返回值永远是一个BIGINT类型的值。

<code>FLOOR(3.8)</code>	→ 3
<code>FLOOR(-3.8)</code>	→ -4

• **LN(x)**

本函数是LOG(x)函数的同义词；最早出现于MySQL 4.0.3。

• **LOG(x)**

LOG( $b, x$ )

返回值：只带一个输入参数的LOG( $x$ )函数将返回 $x$ （以 $e$ 为底）的自然对数。

<code>LOG(0)</code>	→ NULL
<code>LOG(1)</code>	→ 0.000000
<code>LOG(2)</code>	→ 0.693147
<code>LOG(EXP(1))</code>	→ 1.000000



带两个输入参数的 $\text{LOG}(b, x)$ 函数将返回 $x$ 以 $b$ 为底的对数。

```
LOG(10,100)           → 2.0000000
LOG(2,256)            → 8.0000000
```

带两个输入参数的 $\text{LOG}(b, x)$ 函数最早出现于MySQL 4.0.3。在此前的版本里，可以利用公式 $\text{LOG}(x) / \text{LOG}(b)$ 来计算出 $x$ 以 $b$ 为底的对数。

```
LOG(100)/LOG(10)      → 2.000000000
LOG10(100)            → 2.0000000
```

#### • LOG10(x)

返回值： $x$ 以10为底的对数。

```
LOG10(0)              → NULL
LOG10(10)             → 1.0000000
LOG10(100)            → 2.0000000
```

#### • LOG2(x)

返回值： $x$ 以2为底的对数。

```
LOG2(0)               → NULL
LOG2(255)             → 7.994353
LOG2(32767)           → 14.999956
```

$\text{LOG2}()$ 能够让你了解以位计算的数据“宽度”。人们经常利用这个函数来估算某个值的存储空间占用量。

$\text{LOG2}()$ 函数最早出现于MySQL 4.0.3。

#### • MOD(m, n)

返回值：整数除法的余数。 $\text{MOD}(m, n)$ 与“ $m \% n$ ”等价。请参见第C.1.3节。

#### • PI()

返回值：圆周率 $\pi$ 。

```
PI()                  → 3.141593
```

#### • POW(x, y)

返回值： $x^y$ ，即 $x$ 的 $y$ 次方。

```
POW(2,3)              → 8.0000000
POW(2,-3)             → 0.1250000
POW(4,.5)             → 2.0000000
POW(16,.25)           → 2.0000000
```

#### • POWER(x, y)

本函数是 $\text{POW}()$ 函数的同义词。

#### • RADIANS(x)

返回值：角度值 $x$ 的弧度值。

```
RADIANS(0)            → 0
RADIANS(360)          → 6.2831853071796
RADIANS(-360)         → -6.2831853071796
```

- RAND()

- RAND( $n$ )

返回值: RAND()返回一个0.0~1.0之间的浮点随机数。RAND( $n$ )也返回一个0.0~1.0之间的浮点随机数,但它给随机数发生器提供了一个种子值 $n$ ——只要 $n$ 值相同,返回的随机数就相等。可以利用这一特性来生成一组可重复生成的随机数,具体做法是:先以一个给定的 $n$ 值为种子生成一个随机数,然后调用不带参数的RAND()生成一组随机数。当以同样的 $n$ 值为种子重复这一过程时,将得到一组同样的随机数。如下所示:

RAND(10)	→ 0.18109053110805
RAND(10)	→ 0.18109053110805
RAND()	→ 0.7502322306393
RAND()	→ 0.20788959060599
RAND(10)	→ 0.18109053110805
RAND()	→ 0.7502322306393
RAND()	→ 0.20788959060599

请注意例子里带参数和不带参数的RAND()函数的行为特点。

随机数发生器的种子值是针对具体的客户(程序)的。某个客户调用RAND( $n$ )而提供给随机数发生器的种子值不会影响到其他客户调用RAND()函数时得到的随机数。

- ROUND( $x$ )

- ROUND( $x, d$ )

返回值: ROUND( $x$ )的返回值是对 $x$ 进行四舍五入后得到的整数。ROUND( $x, d$ )的返回值是对 $x$ 四舍五入到小数点后面 $d$ 位数后得到的数值;如果 $d$ 等于0,返回值里将不包含小数点和小数部分。

ROUND(15.3)	→ 15
ROUND(15.5)	→ 16
ROUND(-33.27834, 2)	→ -33.28
ROUND(1, 4)	→ 1.0000

ROUND()的具体动作取决于你使用的算术函数库有着怎样的舍入行为。换句话说,ROUND()函数在不同的系统上可能会返回不同的结果。

- SIGN( $x$ )

返回值: 根据 $x$ 是负数、0、正数而分别返回-1、0、1。

SIGN(15.803)	→ 1
SIGN(0)	→ 0
SIGN(-99)	→ -1

- SIN( $x$ )

返回值:  $x$ 的正弦值;  $x$ 被视为是一个弧度值。

SIN(0)	→ 0.000000
SIN(PI())	→ 0.000000
SIN(PI()/2)	→ 1.000000

- **SQRT(*x*)**

返回值：*x*的非负平方根。

SQRT(625)	→ 25.000000
SQRT(2.25)	→ 1.500000
SQRT(-1)	→ NULL

- **TAN(*x*)**

返回值：*x*的正切值；*x*被视为是一个弧度值。

TAN(0)	→ 0.000000
TAN(PI()/4)	→ 1.000000

- **TRUNCATE(*x*, *d*)**

返回值：小数部分被截短为*d*位数字的*x*值。如果*d*等于0，返回值里将不包含小数点和小数部分。如果*d*大于*x*的小数位数，*x*的小数部分将用0来补足到指定的位数。

TRUNCATE(1.23, 1)	→ 1.2
TRUNCATE(1.23, 0)	→ 1
TRUNCATE(1.23, 4)	→ 1.2300

## C.2.4 字符串类函数

绝大多数字符串类函数的返回值仍将是字符串。有些以字符串为输入参数的函数（比如LENGTH()）却会返回一个数值。有相当一部分字符串类的函数是根据字符位置对字符串进行处理的；在介绍这类函数的时候，我们将把字符串的头一个字符（字符串的最左端）称为第一个字符（而不是第0个字符）。

从MySQL 3.23.7开始，下面这些函数都能够支持多字节字符：INSERT()、INSTR()、LCASE()、LEFT()、LOCATE()、LOWER()、MID()、POSITION()、REPLACE()、REVERSE()、RIGHT()、RTRIM()、SUBSTRING()、SUBSTRING\_INDEX()、TRIM()、UCASE()和UPPER()。

- **ASCII(*str*)**

返回值：字符串*str*最左端的那个字符的ASCII编码，如果*str*是一个空字符串，则返回0；如果*str*是NULL，则返回NULL。

ASCII('abc')	→ 97
ASCII('')	→ 0
ASCII(NULL)	→ NULL

- **BIN(*n*)**

返回值：以字符串表示的数值*n*的二进制表示形式。下面两个表达式是等价的：

BIN( <i>n</i> )
CONV( <i>n</i> , 10, 2)

详细情况请参见对CONV()函数的介绍。

BIN()函数最早出现于MySQL 3.22.4。

- **CHAR(*n1*, *n2*, ...)**

返回值：把各输入参数解释为ASCII编码并转换为相应的字符，返回值是这些字符合并在一起而构成的字符串。NULL值将被忽略。

```
CHAR(65)           → 'A'
CHAR(97)           → 'a'
CHAR(89,105,107,101,115,33) → 'Yikes!'
```

- **CHARACTER\_LENGTH(*str*)**

本函数是CHAR\_LENGTH()函数的同义词。

- **CHAR\_LENGTH(*str*)**

返回值：以字符计算的字符串长度。这个函数与LENGTH()差不多，但MySQL 3.23.6及以后版本里的CHAR\_LENGTH()函数具备多字节字符的处理能力：一个多字节字符的长度是1。

- **CHARSET(*str*)**

返回值：字符串*str*所使用的字符集的名字。

```
CHARSET('abc')           → 'latin1'
CHARSET(CONVERT('abc' USING utf8)) → 'utf8'
```

CHARSET()函数最早出现于MySQL 4.1.0。

- **COALESCE(*expr1*, *expr2*, ...)**

返回值：输入参数中的第一个非NULL元素。如果所有元素全都是NULL，则返回NULL。

```
COALESCE(NULL,1/0,2,'a',45+97) → '2'
COALESCE(NULL,1/0)              → NULL
```

COALESCE()函数最早出现于MySQL 3.23.3。

- **CONCAT(*str1*, *str2*, ...)**

返回值：由全体输入参数合并在一起而得到的字符串。只要输入参数中有NULL值，就返回NULL。CONCAT()允许只有一个输入参数的情况。

```
CONCAT('abc','def')           → 'abcdef'
CONCAT('abc')                 → 'abc'
CONCAT('abc',NULL)            → NULL
CONCAT('Hello',' ',' ','goodbye') → 'Hello, goodbye'
```

如果CONCAT()函数的输入参数使用了不同的字符集（从MySQL 4.1开始，就有可能出现这种情况），返回值将使用第一个输入参数的字符集。

合并字符串的另一种方法是相邻指定它们，如下所示：

```
'three' 'blind' 'mice'       → 'threeblindmice'
'abc' 'def' = 'abcdef'       → 1
```

- **CONCAT\_WS(*delim*, *str1*, *str2*, ...)**

返回值：由第2个及后续输入参数合并在一起而得到的字符串，各输入参数之间用字符串*delim*加以分隔。如果*delim*是NULL，则返回NULL；但参加合并的字符串里的NULL值和空字符串都将被忽略。

```
CONCAT_WS(',', 'a', 'b', 'c', 'd') → 'a,b,c,d'
CONCAT_WS('*-*-', 'lemon', '', 'lime', NULL, 'grape') → 'lemon*-*lime*-*grape'
```

#### • CONV(*n*, *from\_base*, *to\_base*)

返回值：把以*from\_base*为底（即*from\_base*进制）的数值*n*转换为以*to\_base*为底（即*to\_base*进制），并把转换结果表示为一个字符串。只要参数中有NULL值，就将返回NULL。参数*from\_base*和*to\_base*必须是一个2~36之间的整数。*n*将被看做是一个BIGINT（64位）整数值，但不能被给定为一个字符串——因为底数大于10的数值可能会包含着用来充当数字的部分字母（这也是我们可能会在CONV()函数对底数是从11~36之间某个数字的数值进行转换而返回的字符串里看到一些从A~Z的字母的原因）。如果*n*不是一个合法的*from\_base*进制数值，CONV()函数的返回值就将是0。（比如说，如果*from\_base*等于16而*n*是'abcdefg'，那么，因为g不是一个合法的十六进制数字，CONV()函数的返回值就将是0）。

数值*n*里的非十进制数字既可以是大大写字母，也可以是小写字母；但返回值里的非十进制数字将全部为大写字母。

下面这个例子将把十六进制数e（即十进制数14）转换为二进制：

```
CONV('e', 16, 2) → '1110'
```

下面这个例子将把二进制数11111111（即十进制数255）转换为八进制：

```
CONV(11111111, 2, 8) → '377'
CONV('11111111', 2, 8) → '377'
```

在默认的情况下，数值*n*将被视为一个无符号数。但如果你给出的*to\_base*是一个负值，函数CONV()就将把*n*视为一个带正负符号的数值。如下所示：

```
CONV(-10, 10, 16) → 'FFFFFFFFFFFFFFF6'
CONV(-10, 10, -16) → '-A'
```

CONV()函数最早出现于MySQL 3.22.4。

#### • ELT(*n*, *str1*, *str2*, ...)

返回值：*str1*, *str2*, ...等字符串中的第*n*个字符串。当*n*是NULL、第*n*个字符串是NULL或者不存在第*n*个字符串的时候，返回NULL。所谓第1个字符串指的是*str1*。ELT()函数与后面介绍的FIELD()函数互为补充。

```
ELT(3, 'a', 'b', 'c', 'd', 'e') → 'c'
ELT(0, 'a', 'b', 'c', 'd', 'e') → NULL
ELT(6, 'a', 'b', 'c', 'd', 'e') → NULL
ELT(FIELD('b', 'a', 'b', 'c'), 'a', 'b', 'c') → 'b'
```

#### • EXPORT\_SET(*n*, *on*, *off*, [*delim*, [*bit\_count*]])

返回值：返回一个由字符串*on*和*off*构成、以字符串*delim*为分隔符的字符串；每个*on*对应着整数*n*中一个被置位（即等于1）的位，每个*off*对应着整数*n*中一个没有被置位（即等于0）的位。字符串*on*和*off*在返回值中的排列顺序正好与它们所对应的*n*值的各个位的排列顺序相反。*bit\_count*是将对*n*值进行如此转换的最大位数，如果*bit\_count*大于*n*值的位数，



则在返回值中用字符串 *off* 补足。如果 *delim* 默认, 则使用逗号作为分隔符; *bit\_count* 的默认值是64。只要输入参数中有NULL值, 这个函数的返回值就将是NULL。

```
EXPORT_SET(7, '+', '-', '', 5)           → '++--'
EXPORT_SET(0xa, '1', '0', '', 6)         → '010100'
EXPORT_SET(97, 'Y', 'N', '', 8)          → 'Y,N,N,N,N,Y,Y,N'
```

EXPORT\_SET()函数最早出现于MySQL 3.23.2。

#### • FIELD(*str*, *str1*, *str2*, ...)

返回值: 在 *str1*, *str2*, ... 中找到 *str* 并返回匹配字符串的下标; 如果没有找到匹配或者 *str* 是NULL值时, 返回NULL。字符串 *str1* 的下标是1。FIELD()函数与前面介绍的ELT()函数互为补充。

```
FIELD('b', 'a', 'b', 'c')                → 2
FIELD('d', 'a', 'b', 'c')                → 0
FIELD(NULL, 'a', 'b', 'c')               → 0
FIELD(ELT(2, 'a', 'b', 'c'), 'a', 'b', 'c') → 2
```

#### • FIND\_IN\_SET(*str*, *str\_list*)

返回值: *str\_list* 是由一些以逗号分隔的子串 (即类似于MySQL中的一个SET值) 构成的字符串。FIND\_IN\_SET()将返回字符串 *str* 在 *str\_list* 中的下标。如果 *str* 没有出现在 *str\_list* 里, 返回0; 只要输入参数中有NULL值, 就返回NULL。第一个子串的下标是1。

```
FIND_IN_SET('cow', 'moose,cow,pig')       → 2
FIND_IN_SET('dog', 'moose,cow,pig')       → 0
```

#### • FORMAT(*x*, *d*)

返回值: 把数值 *n* 舍入到小数点后面第 *d* 位数字并写成 *nn,nnn.nnn* 的格式, 返回值是一个字符串。如果 *d* 等于0, 则返回值中将不包含小数点和小数部分。

```
FORMAT(1234.56789, 3)                    → '1,234.568'
FORMAT(999999.99, 2)                     → '999,999.99'
FORMAT(999999.99, 0)                     → '1,000,000'
```

请注意最后一个例子中的数值舍入行为。

#### • HEX(*n*)

HEX(*str*)

返回值: 如果输入参数是一个数值 *n*, HEX()函数将返回 *n* 的十六进制表示形式, 返回值是一个字符串。下面两个表达式是等价的:

```
HEX(n)
CONV(n, 10, 16)
```

详细情况请参见对CONV()函数的介绍。

在MySQL 4.0.1之前的版本里, HEX()函数总是把自己的输入参数解释为一个字符串:

```
HEX(255)                                → 'FF'
HEX('255')                             → 'FF'
```

但在MySQL 4.0.1及以后的版本里，如果HEX()函数的输入参数是一个字符串`str`，就将把`str`中的每一个字符转换为一个两位数的十六进制数，并把`str`完整的转换结果返回为一个字符串：

```
HEX('255')           → '323535'
HEX('abc')           → '616263'
```

HEX()函数最早出现于MySQL 3.22.4。

• **INSERT(*str*, *pos*, *len*, *ins\_str*)**

返回值：把字符串`str`从第`pos`个位置开始的`len`个字符替换为`ins_str`后得到的一个字符串。如果`pos`超出字符串`str`的长度范围，则返回原来的字符串`str`；只要输入参数中有NULL值，就返回NULL。

```
INSERT('nighttime',6,4,'fall')      → 'nightfall'
INSERT('sunshine',1,3,'rain or ')   → 'rain or shine'
INSERT('sunshine',0,3,'rain or ')   → 'sunshine'
```

• **INSTR(*str*, *substr*)**

返回值：INSTR()类似于带两个输入参数（但顺序却前后颠倒）的LOCATE()函数。也就是说，下面两个表达式是等价的：

```
INSTR(str,substr)
LOCATE(substr,str)
```

• **LCASE(*str*)**

本函数是LOWER()函数的同义词。

• **LEFT(*str*, *len*)**

返回值：字符串`str`最左面的`len`个字符，如果`len`大于`str`的长度，则返回整个字符串`str`。如果`str`是NULL，则返回NULL；如果`len`是NULL或者小于1，则返回一个空字符串。

```
LEFT('my left foot', 2)           → 'my'
LEFT(NULL,10)                     → NULL
LEFT('abc',NULL)                  → ''
LEFT('abc',0)                     → ''
```

• **LENGTH(*str*)**

返回值：字符串`str`的长度。

```
LENGTH('abc')                    → 3
LENGTH('')                        → 0
LENGTH(NULL)                      → NULL
```

• **LOCATE(*substr*, *str*)**

LOCATE(*substr*, *str*, *pos*)

返回值：只带两个输入参数的LOCATE()函数将返回子串`substr`在字符串`str`里第一次出现的位置；如果子串`substr`没有出现在字符串`str`里，则返回0。只要输入参数中有NULL值，就返回NULL。如果还给出了一个位置参数`pos`，LOCATE()函数将在字符串`str`里以`pos`为

起点去寻找子串 *substr*。在MySQL 4.0.1及以后的版本里，如果 *str* 和 *substr* 都不是二进制字符串，比较操作将不区分字母的大小写情况。

```
LOCATE('b','abc')           → 2
LOCATE('b','ABC')           → 2
LOCATE(BINARY 'b','ABC')    → 0
```

在MySQL 4.0.1之前的版本里，LOCATE()函数里的比较操作是区分字母大小写情况的：

```
LOCATE('b','abc')           → 2
LOCATE('b','ABC')           → 0
```

#### • LOWER(*str*)

返回值：把字符串 *str* 里的字符全都转换为小写字母后得到的字符串。如果 *str* 是 NULL，则返回 NULL。

```
LOWER('New York, NY')      → 'new york, ny'
LOWER(NULL)                 → NULL
```

#### • LPAD(*str*, *len*, *pad\_str*)

返回值：在字符串 *str* 的左侧用子串 *pad\_str* 补足到长度等于 *len* 个字符时得到的字符串。只要输入参数中有 NULL 值，就返回 NULL。

```
LPAD('abc',12,'def')       → 'defdefdefabc'
LPAD('abc',10,'.')          → '.....abc'
```

在MySQL 3.23.29及以后的版本里，如果字符串 *str* 的长度已经超过 *len* 个字符，LPAD() 将把字符串 *str* 截短为 *len* 个字符：

```
LPAD('abc',2,'.')          → 'ab'
```

在MySQL 3.23.29之前的版本里，如果字符串 *str* 的长度已经超过 *len* 个字符，LPAD() 将原样返回字符串 *str*：

```
LPAD('abc',2,'.')          → 'abc'
```

LPAD()函数最早出现于MySQL 3.22.2。

#### • LTRIM(*str*)

返回值：去掉字符串 *str* 最左端的一个字符后得到的字符串。如果 *str* 是 NULL，则返回 NULL。

```
LTRIM(' abc ')             → 'abc'
```

#### • MAKE\_SET(*n*, *bit0\_str*, *bit1\_str*, ...)

返回值：根据整数值 *n* 和子串 *bit0\_str*, *bit1\_str*, ... 而构造出来的一个 SET 值（即一个以逗号来分隔各个子串的字符串）。*n* 值中每一个被置位（即等于 1）的位所对应的子串都将被包括在返回值里。（比如说，如果 *n* 值的位 0 被置位，*bit0\_str* 就会被包括在结果里。）如果 *n* 等于 0，则返回一个空字符串；如果 *n* 是 NULL，则返回 NULL。子串 *bit0\_str*, *bit1\_str*, ... 中的 NULL 将在构造结果字符串时被忽略。

MAKE_SET(8, 'a', 'b', 'c', 'd', 'e')	→ 'd'
MAKE_SET(7, 'a', 'b', 'c', 'd', 'e')	→ 'a,b,c'
MAKE_SET(2+16, 'a', 'b', 'c', 'd', 'e')	→ 'b,e'
MAKE_SET(2 16, 'a', 'b', 'c', 'd', 'e')	→ 'b,e'
MAKE_SET(-1, 'a', 'b', 'c', 'd', 'e')	→ 'a,b,c,d,e'

在最后一个例子里，因为 $n$ 等于-1，所以返回值将包含每一个子串。

MAKE\_SET()函数最早出现于MySQL 3.22.2。

#### • MATCH(column\_list) AGAINST(str)

MATCH(column\_list) AGAINST(str IN BOOLEAN MODE)

MATCH()将使用一个FULLTEXT索引来进行一次搜索操作。*column\_list*是由一个或者多个数据列的名字构成的列表，各数据列的名字以逗号分隔；而将被搜索的数据表里也必须有一个由这些数据列构成的FULLTEXT索引。AGAINST(*str*)里的*str*是你想在这些数据列里查找的一个或者多个单词，单词是由字母、数字、单引号或者下划线字符构成的字符序列。MATCH(*column\_list*)中的*column\_list*部分允许出现括号，但AGAINST(*str*)中的*str*部分不允许出现括号。

MATCH给出的是被搜索单词在每个数据行里的出现次数分布统计值。这些分布统计值都是非负的浮点数：零分布统计值表示在数据表里没有找到被搜索单词，正分布统计值则表明至少有一个被搜索单词在数据表里被找到过。如果被搜索单词在数据表一半以上的数据行里都被找到过，它们的分布情况将被认为是零——因为它们的出现次数太多了。此外，MySQL内部还有一个停止单词（比如“the”和“but”）清单，这个清单里的单词即使被你用作搜索单词，MySQL也不会对它们进行搜索——因为它们的出现次数往往会太多了。如果AGAINST(*str*)部分还带有“IN BOOLEAN MODE”，那么搜索结果将以被搜索单词是出现过还是没有出现过为依据，而不是以它们的出现频度为依据。在AGAINST(*str* IN BOOLEAN MODE)搜索方式中，还可以通过给*str*中的被搜索单词加上以下几种修饰符的办法来影响搜索操作的具体行为：

##### • + 或 -

出现在被搜索单词前面的加号(+)或减号(-)表示该单词必须出现或必须不出现。

##### • < 或 >

出现在被搜索单词前面的小于号(<)或大于号(>)将削弱或增加该单词对分布统计值计算结果的贡献。

##### • ~

出现在被搜索单词前面的波浪号(~)将完全取消该单词对分布统计值计算结果的贡献，但如果这个单词占据了整个数据行，则仍对分布统计值计算结果有贡献——这与减号(-)修饰符的功用是有差异的。

##### • \*

出现在被搜索单词尾部的星号字符(\*)被看做是一个通配符。比如说，“act\*”将匹配act、acts、action等等。

- "phrase"

如果被搜索单词是一个包含有空格的短语,就必须用双引号把它引起来。"phrase"形式的短语搜索必须在有关单词的排列顺序也完全一致时才算匹配成功。

- ()

多个被搜索单词可以用括号归组为一个表达式。

在AGAINST(*str* IN BOOLEAN MODE)搜索方式中,那些不带修饰符的被搜索单词都是可选的,与普通搜索方式(即非AGAINST(*str* IN BOOLEAN MODE)搜索方式)中的含义相同。

AGAINST(*str* IN BOOLEAN MODE)搜索方式在数据表没有相应的FULLTEXT索引时也能进行,但速度相当慢。

FULLTEXT搜索机制最早出现于MySQL 3.23.23, AGAINST(*str* IN BOOLEAN MODE)搜索机制最早出现于MySQL 4.0.1, 短语搜索机制最早出现于MySQL 4.0.2。

有关FULLTEXT搜索机制的更多信息请参见第3章。

- MID(*str*, *pos*, *len*)

MID(*str*, *pos*)

返回值: MID(*str*, *pos*, *len*)将返回字符串*str*从位置*pos*开始且长度为*len*个字符的子串。

MID(*str*, *pos*)将返回字符串*str*从位置*pos*到最后一个字符串的子串。只要输入参数中有NULL值,就返回NULL。

```
MID('what a dull example',8,4)           → 'dull'
MID('what a dull example',8)             → 'dull example'
```

事实上, MID()是SUBSTRING()函数的同义词。SUBSTRING()函数的各种语法形式都能用在MID()函数里。

- OCT(*n*)

返回值: 数值*n*的八进制表示形式, 返回值是一个字符串。下面两个表达式是等价的:

```
OCT(n)
CONV(n,10,8)
```

详细情况请参见对CONV()函数的介绍。

OCT()函数最早出现于MySQL 3.22.4。

- OCTET\_LENGTH(*str*)

本函数是LENGTH()函数的同义词。

- POSITION(*substr* IN *str*)

本函数相当于只带两个输入参数的LOCATE()函数。下面两个表达式是等价的:

```
POSITION(substr IN str)
LOCATE(substr,str)
```

- ORD(*str*)

返回值: 字符串*str*第一个字符的排位序号, 如果*str*是NULL, 则返回NULL。如果第一个字符不是一个多字节字符, ORD()函数将等价于ASCII()函数。



```
ORD('abc') → 97
ASCII('abc') → 97
```

如果字符串`str`的第一个字符是一个多字节字符，ORD()函数将按以下公式依次累加其各个字节(`b1`到`bn`)的ASCII编码值，如下所示：

$$(\dots ((\text{ASCII}(\mathbf{b1}) * 256 + \text{ASCII}(\mathbf{b2})) * 256 + \dots) + \text{ASCII}(\mathbf{bn}))$$

ORD()函数最早出现于MySQL 3.23.6。

#### • QUOTE(`str`)

返回值：按SQL语句的使用要求在输入参数`str`里正确地添加各种引号后得到的字符串。这个函数在编写能动态生成其他查询语句的查询语句时非常有用。对于非NULL值的输入参数`str`，QUOTE()将给其中的每一个单引号、ASCII NUL、反斜线和Ctrl-Z字符加上一个反斜线(\)字符以进行转义，并用单引号把转换后得到的字符串引起来。对于NULL值输入参数`str`，QUOTE()的返回值是不带单引号的单词“NULL”。如下所示：

```
QUOTE('X') → 'X'
QUOTE('') → '\ '
QUOTE(NULL) → NULL
```

QUOTE()函数最早出现于MySQL 4.0.3。

#### • REPEAT(`str`, `n`)

返回值：把字符串`str`重复`n`次后得到的字符串。如果`n`是负数或零，则返回一个空字符串。只要输入参数中有NULL值，就返回NULL。

```
REPEAT('x', 10) → 'xxxxxxxxxx'
REPEAT('abc', 3) → 'abccabccabc'
```

#### • REPLACE(`str`, `from_str`, `to_str`)

返回值：把字符串`str`中的子串`from_str`全部替换为`to_str`后得到的字符串。如果`to_str`是空字符串，则效果相当于把`str`中的子串`from_str`全都去掉。如果`from_str`是空字符串，REPLACE()将不对字符串`str`做任何改变。只要输入参数中有NULL值，就返回NULL。

```
REPLACE('abracadabra', 'a', 'oh') → 'ohbrohcohdohbroh'
REPLACE('abracadabra', 'a', '') → 'brcdbr'
REPLACE('abracadabra', '', 'x') → 'abracadabra'
```

#### • REVERSE(`str`)

返回值：前后颠倒字符串`str`里的所有字符后得到的字符串。若`str`是NULL，则返回NULL。

```
REVERSE('abracadabra') → 'arbadacarba'
REVERSE('tararA ta tar a raT') → 'Tar a rat at Ararat'
```

#### • RIGHT(`str`, `len`)

返回值：字符串`str`最右面的`len`个字符，如果`len`大于`str`的长度，则返回整个字符串`str`。如果`str`是NULL，则返回NULL；如果`len`是NULL或者小于1，则返回一个空字符串。

```
RIGHT('rightmost', 4) → 'most'
```

- **RPAD(str, len, pad\_str)**

返回值：字符串 *str* 的值组成的字符串，右边用字符串 *pad\_str* 补到 *len* 个字符长。只要输入参数中有 NULL 值，就返回 NULL。

```
RPAD('abc', 12, 'def')      → 'abcdefdefdef'
RPAD('abc', 10, '.')        → 'abc.....'
```

在 MySQL 3.23.29 及以后的版本里，如果字符串 *str* 的长度已经超过 *len* 个字符，LPAD() 将把字符串 *str* 截短为 *len* 个字符：

```
RPAD('abc', 2, '.')        → 'ab'
```

在 MySQL 3.23.29 之前的版本里，如果字符串 *str* 的长度已经超过 *len* 个字符，LPAD() 将原样返回字符串 *str*：

```
RPAD('abc', 2, '.')        → 'abc'
```

RPAD() 函数最早出现于 MySQL 3.22.2。

- **RTRIM(str)**

返回值：去掉字符串 *str* 最右端的一个字符后得到的字符串。如果 *str* 是 NULL，则返回 NULL。

```
RTRIM(' abc ')             → ' abc'
```

- **SOUNDEX(str)**

返回值：根据字符串 *str* 计算出来的 soundex 字符串，如果 *str* 是 NULL，则返回 NULL。字符串 *str* 中不是字母或数字的字符都将被忽略。不在从 A~Z 范围以内的非字母国际字符都将被视为元音。如下所示：

```
SOUNDEX('Cow')              → 'C000'
SOUNDEX('Cowl')              → 'C400'
SOUNDEX('Howl')              → 'H400'
SOUNDEX('Hello')             → 'H400'
```

- **SPACE(n)**

返回值：一个由 *n* 个空格构成的字符串。如果 *n* 不是一个正数，则返回一个空字符串；如果 *n* 是 NULL，则返回 NULL。

```
SPACE(6)                    → '      '
SPACE(0)                    → ' '
SPACE(NULL)                  → NULL
```

- **SUBSTRING(str, pos)**

**SUBSTRING(str, pos, len)**

**SUBSTRING(str FROM pos)**

**SUBSTRING(str FROM pos FOR len)**

返回值：字符串 *str* 从位置 *pos* 开始的一个子串。只要输入参数中有 NULL 值，就返回 NULL。如果给出了 *len* 参数，则作为返回值子串的长度将是 *len* 个字符；否则，将返回字符串 *str* 从位置 *pos* 开始直到最后一个字符的子串。

```
SUBSTRING('abcdef', 3)      → 'cdef'
SUBSTRING('abcdef', 3, 2)    → 'cd'
```

下面这几个表达式都是等价的：

```
SUBSTRING(str,pos,len)
SUBSTRING(str FROM pos FOR len)
MID(str,pos,len)
```

• SUBSTRING\_INDEX(str, delim, n)

返回值：按以下规则得到的字符串 *str* 的一个子串：1) 如果 *n* 是正值，SUBSTRING\_INDEX() 函数将按从左向右的顺序找到子串 *delim* 的第 *n* 次出现并返回该位置左侧的全部内容；2) 如果 *n* 是负值，SUBSTRING\_INDEX() 函数将按从右向左的顺序找到子串 *delim* 的第 *n* 次出现并返回该位置右侧的全部内容；3) 如果没有在字符串 *str* 里找到子串 *delim*，则原样返回字符串 *str*；4) 只要输入参数中有 NULL 值，就返回 NULL。

```
SUBSTRING_INDEX('jar-jar','j',-2)      → 'ar-jar'
SUBSTRING_INDEX('sampadm@localhost','@',1) → 'sampadm'
SUBSTRING_INDEX('sampadm@localhost','@',-1) → 'localhost'
```

• TRIM([ LEADING | TRAILING | BOTH ] [ trim\_str ] FROM ] str)

返回值：按以下规则在字符串 *str* 的首/尾去掉子串 *trim\_str* 后得到的字符串：1) 如果给出了关键字 LEADING，TRIM() 函数将去掉字符串 *str* 最前（左）端的子串 *trim\_str*；2) 如果给出了关键字 TRAILING，TRIM() 函数将去掉字符串 *str* 最尾（右）端的子串 *trim\_str*；3) 如果给出了关键字 BOTH，TRIM() 函数将去掉字符串 *str* 前后（左、右）两端的子串 *trim\_str*；4) 如果 LEADING、TRAILING 或 BOTH 都没有给出，则按 BOTH 情况做默认处理；5) 如果没有给定子串 *trim\_str*，TRIM() 函数将去掉字符串 *str* 首/尾的空格；6) 连续出现的子串 *trim\_str* 将都被去掉。如下所示：

```
TRIM('^' FROM '^xyz^')      → 'xyz'
TRIM(LEADING '^' FROM '^xyz^') → 'xyz^'
TRIM(TRAILING '^' FROM '^xyz^') → '^xyz'
TRIM(BOTH '^' FROM '^xyz^') → 'xyz'
TRIM(BOTH FROM '  abc  ') → 'abc'
TRIM('  abc  ') → 'abc'
```

• UCASE(str)

本函数是 UPPER() 函数的同义词。

• UPPER(str)

返回值：把字符串 *str* 里的字符全都转换为大写字母后得到的字符串。如果 *str* 是 NULL，则返回 NULL。

```
UPPER('New York, NY')      → 'NEW YORK, NY'
UPPER(NULL)                → NULL
```

## C.2.5 日期和时间类函数

日期和时间类函数允许输入参数是多种类型。一般说来，接受 DATE 值作为其输入参数的函数通常也接受 DATETIME 或 TIMESTAMP 值作为其参数并将忽略其中的时间部分；而接受 TIME 值作为其输入参数的函数通常也接受 DATETIME 或 TIMESTAMP 值作为其输入参数并将忽略其

中的日期部分。

本节中的大部分函数都能把数值形式的输入参数解释为日期和时间值，如下所示：

```
MONTH('2004-07-25')      → 7
MONTH(20040725)           → 7
```

类似地，在数值上下文里，那些返回值原本是日期和时间值的函数也大都能返回一个数值，如下所示：

```
CURDATE()                → '2002-05-14'
CURDATE() + 0             → 20020514
```

如果传递给日期和时间类函数的参数不是合法值，其返回值将难以预料。因此，在调用日期和时间类函数之前，请一定要把好输入参数这一关。

- **ADDDATE(*date*, INTERVAL *expr interval*)**

本函数是DATE\_ADD()函数的同义词。

- **CURDATE()**

返回值：当前日期。根据该函数被调用时的上下文，返回值或者是一个'CCYY-MM-DD'格式的字符串，或者是一个CCYYMMDD格式的数值。

```
CURDATE()                → '2002-05-14'
CURDATE() + 0             → 20020514
```

- **CURRENT\_DATE()**

本函数是CURDATE()函数的同义词，且括号可选。

- **CURRENT\_TIME()**

本函数是CURTIME()函数的同义词，且括号可选。

- **CURRENT\_TIMESTAMP()**

本函数是NOW()函数的同义词，且括号可选。

- **CURTIME()**

返回值：当前时间。根据该函数被调用时的上下文，返回值或者是一个'hh:mm:ss'格式的字符串，或者是一个hhmmss格式的数值。

```
CURTIME()                → '09:51:36'
CURTIME() + 0             → 95136
```

- **DATE\_ADD(*date*, INTERVAL *expr interval*)**

返回值：日期和时间值*date*加上（若表达式*expr*以负号“-”开头，则减去）一段时间间隔后得到的日期和时间值。其中，表达式*expr*给出了将与*date*值进行加减的数值，时间类型标识符*interval*则表明了这段时间间隔的解释办法。如果*date*是一个DATE值，则返回值也将是一个DATE值，有关参数中的时间部分不参加计算；否则，返回值将是一个DATETIME值。如果*date*不是一个合法的日期和时间值，DATE\_ADD()函数将返回NULL。

```
DATE_ADD('2002-12-01', INTERVAL 1 YEAR)      → '2003-12-01'
DATE_ADD('2002-12-01', INTERVAL 60 DAY)      → '2003-01-30'
DATE_ADD('2002-12-01', INTERVAL -3 MONTH)    → '2002-09-01'
DATE_ADD('2002-12-01 08:30:00', INTERVAL 12 HOUR) → '2002-12-01 20:30:00'
```

表C-3列出了时间类型标识符`interval`的可取值、含义以及使用格式。关键字INTERVAL和具体的时间类型标识符（即标识符`interval`的取值）允许以任意大小写字母形式给出。

表C-3 DATE\_ADD()函数的时间类型标识符

时间类型标识符	含 义	输入参数 <code>expr</code> 的格式
SECOND	秒	<code>ss</code>
MINUTE	分钟	<code>mm</code>
HOUR	小时	<code>hh</code>
DAY	天	<code>DD</code>
MONTH	月	<code>MM</code>
YEAR	年	<code>YY</code>
MINUTE_SECOND	分钟和秒	<code>'mm:ss'</code>
HOUR_MINUTE	小时和分钟	<code>'hh:mm'</code>
HOUR_SECOND	小时、分钟和秒	<code>'hh:mm:ss'</code>
DAY_HOUR	天和小时	<code>'DD hh'</code>
DAY_MINUTE	天、小时和分钟	<code>'DD hh:mm'</code>
DAY_SECOND	天、小时、分钟和秒	<code>'DD hh:mm:ss'</code>
YEAR_MONTH	年和月	<code>'YY-MM'</code>

表达式`expr`既可以给定为一个数值，也可以给定为一个字符串；但如果其中包含有非数字的字符，那就必须给定为一个字符串。表达式`expr`中的分隔符允许是任何一种标点符号。

`DATE_ADD('2002-12-01', INTERVAL '2:3' YEAR_MONTH) → '2005-03-01'`

`DATE_ADD('2002-12-01', INTERVAL '2-3' YEAR_MONTH) → '2005-03-01'`

表达式`expr`的各组成部分将按时间类型标识符`interval`限定的输入格式从右向左进行匹配和解释。比如说，时间类型标识符HOUR\_SECOND限定的输入格式是`'hh:mm:ss'`，如果表达式`expr`的值是`'15:21'`，那它将被解释为`'00:15:21'`而不是`'15:21:00'`。

`DATE_ADD('2002-12-01 12:00:00', INTERVAL '15:21' HOUR_SECOND)`  
`→ '2002-12-01 12:15:21'`

如果时间类型标识符`interval`是YEAR、MONTH或YEAR\_MONTH，而计算结果中的天数部分却大于当月的天数，DATE\_ADD()函数将把天数自动调整为当月的最大日份值，如下所示：

`DATE_ADD('2002-12-31', INTERVAL 2 MONTH) → '2003-02-28'`

DATE\_ADD()函数最早出现于MySQL 3.22.4。此外，MySQL 3.23.4及以后的版本还支持使用下面这种语法格式：

`'2002-12-31' + INTERVAL 2 MONTH → '2003-02-28'`

`INTERVAL 2 MONTH + '2002-12-31' → '2003-02-28'`

#### • DATE\_FORMAT(date, format)

返回值：按格式字符串`format`对日期和时间值`date`进行格式编排后得到的字符串。这个函数能把MySQL所支持的各种DATE和DATETIME格式重新编排为给定的格式。

`DATE_FORMAT('2002-12-01', '%M %e, %Y')` `→ 'December 1, 2002'`

`DATE_FORMAT('2002-12-01', 'The %D of %M')` `→ 'The 1st of December'`



表C-4列出了可以用在格式字符串`format`里的各种格式标识符。

表C-4 DATE\_FORMAT()函数的格式标识符

格式标识符	含 义
%S, %s	以两位数字表示的秒值 (00, 01, ..., 59)
%i	以两位数字表示的分钟值 (00, 01, ..., 59)
%H	以两位数字表示的小时值, 24小时制 (00, 01, ..., 23)
%h, %l	以两位数字表示的小时值, 12小时制 (00, 01, ..., 12)
%k	以数值表示的小时值, 24小时制 (0, 1, ..., 23)
%l	以数值表示的小时值, 12小时制 (0, 1, ..., 12)
%T	24小时制的时间值 (hh:mm:ss)
%r	12小时制的时间值 (hh:mm:ss AM或者hh:mm:ss PM)
%p	AM或者PM
%W	星期几 (Sunday, Monday, ..., Saturday)
%a	星期几的简写形式 (Sun, Mon, ..., Sat)
%d	以两位数字表示的日 (00, 01, ..., 31)
%e	以数值表示的日 (1, 2, ..., 31)
%D	英语后缀方式的日 (1st, 2nd, 3rd, ...)
%w	以数值表示的星期几 (0=Sunday, 1=Monday, ..., 6=Saturday)
%j	以三位数字表示的一年中的天数 (001, 002, ..., 366)
%U	一年中的第几个星期 (00, ..., 53), 以Sunday (星期日) 作为每星期的第一天
%u	一年中的第几个星期 (00, ..., 53), 以Monday (星期一) 作为每星期的第一天
%V	一年中的第几个星期 (01, ..., 53), 以Sunday (星期日) 作为每星期的第一天
%v	一年中的第几个星期 (01, ..., 53), 以Monday (星期一) 作为每星期的第一天
%M	月份值 (January, February, ..., December)
%b	月份值的简写形式 (Jan, Feb, ..., Dec)
%m	以两位数字表示的月份值 (01, 02, ..., 12)
%c	以数值表示的月份值 (1, 2, ..., 12)
%Y	以四位数字表示的年份值
%y	以两位数字表示的年份值
%X	以Sunday (星期日) 作为每星期第一天的年份值, 以四位数字表示
%x	以Monday (星期一) 作为每星期第一天的年份值, 以四位数字表示
%%	%字符本身

注意, 格式标识符前面的百分号 (%) 不允许省略。(MySQL 3.23之前的版本虽然允许使用 “%”, 但并不要求它必须被写出。) 此外, 格式字符串里不是格式标识符的字符都将被原样复制到结果字符串里去。

如果你为DATE值设定了一个带时间部分的格式字符串, 那它的时间部分将取值为 '00:00:00'。如下所示:

```
DATE_FORMAT('2002-12-01', '%i') → '00'
```

格式标识符 %v、%V、%x、%X 最早出现于 MySQL 3.23.8。

#### • DATE\_SUB(date, INTERVAL expr interval)

返回值: 按与DATE\_ADD()函数同样的规则计算出来的日期和时间值。只不过DATE\_SUB()函数是对表达式`expr`和日期和时间值`date`做减法。详细情况请参见DATE\_ADD()条目。

```

DATE_SUB('2002-12-01',INTERVAL 1 MONTH)           → '2002-11-01'
DATE_SUB('2002-12-01',INTERVAL '13-2' YEAR_MONTH) → '1989-10-01'
DATE_SUB('2002-12-01 04:53:12',INTERVAL '13-2' MINUTE_SECOND)
                                                    → '2002-12-01 04:40:10'
DATE_SUB('2002-12-01 04:53:12',INTERVAL '13-2' HOUR_MINUTE)
                                                    → '2002-11-30 15:51:12'

```

DATE\_SUB()函数最早出现于MySQL 3.22.4。此外,MySQL 3.23.4及以后的版本还支持使用下面这种语法格式:

```
'2002-12-01' - INTERVAL 1 MONTH           → '2002-11-01'
```

#### • DAYNAME(date)

返回值:日期和时间值`date`是星期几,返回值是字符串形式的星期几。

```

DAYNAME('2002-12-01')           → 'Sunday'
DAYNAME('1900-12-01')           → 'Saturday'

```

#### • DAYOFMONTH(date)

返回值:日期和时间值`date`是几日,返回值是字符串形式的几日。

```

DAYOFMONTH('2002-12-01')        → 1
DAYOFMONTH('2002-12-25')        → 25

```

#### • DAYOFWEEK(date)

返回值:日期和时间值`date`是星期几,返回值是数值形式的星期几。按照ODBC标准的规定,星期几的范围将是1~7,其中1代表Sunday(星期日)、7代表Saturday(星期六)。另请参见WEEKDAY()条目。

```

DAYOFWEEK('2002-12-08')        → 1
DAYNAME('2002-12-08')          → 'Sunday'
DAYOFWEEK('2002-12-14')        → 7
DAYNAME('2002-12-14')          → 'Saturday'

```

#### • DAYOFYEAR(date)

返回值:日期和时间值`date`是一年中的第几天,取值范围是1~366。

```

DAYOFYEAR('2002-12-01')        → 335
DAYOFYEAR('2004-12-31')        → 366

```

#### • EXTRACT(interval FROM datetime)

返回值:从日期和时间值`datetime`里根据时间类型标识符`interval`(允许是表C-3中的任何一个)而截取出来的部分。

```

EXTRACT(YEAR FROM '2002-12-01 13:42:19')      → 2002
EXTRACT(MONTH FROM '2002-12-01 13:42:19')     → 12
EXTRACT(DAY FROM '2002-12-01 13:42:19')       → 1
EXTRACT(HOUR_MINUTE FROM '2002-12-01 13:42:19') → 1342
EXTRACT(SECOND FROM '2002-12-01 13:42:19')    → 19

```

从MySQL 3.23.39开始,EXTRACT()函数还能对有“残缺”的日期和时间值进行截取:

```

EXTRACT(YEAR FROM '2004-00-12')      → 2004
EXTRACT(MONTH FROM '2004-00-12')     → 0
EXTRACT(DAY FROM '2004-00-12')       → 12

```

EXTRACT()函数最早出现于MySQL 3.23.0。

#### • FROM\_DAYS(*n*)

返回值：把从公元0年1月1日开始计算的天数*n*（通常是一个来自TO\_DAYS()函数的返回值）转换为相应的日期，它是一个字符串。

```

TO_DAYS('2009-12-01')                → 734107
FROM_DAYS(734107 + 3)                  → '2009-12-04'

```

FROM\_DAYS()函数只能转换出格里高里历法日期（始于公元1582年）。

#### • FROM\_UNIXTIME(*unix\_timestamp*)

FROM\_UNIXTIME(*unix\_timestamp*, *format*)

返回值：根据该函数被调用时的上下文，返回值或者是一个'*CCYY-MM-DD hh:mm:ss*'格式的字符串，或者是一个'*CCYYMMDDhhmmss*'格式的数值；*unix\_timestamp*是一个UNIX时间戳值（通常是一个来自UNIX\_TIMESTAMP()函数的返回值）。如果还给出了格式字符串*format*，返回值就将按该字符串设定的格式编排，就像DATE\_FORMAT()函数一样。

```

UNIX_TIMESTAMP()                      → 1021389416
FROM_UNIXTIME(1021389416)              → '2002-05-14 10:16:56'
FROM_UNIXTIME(1021389416, '%Y')       → '2002'

```

#### • HOUR(*time*)

返回值：日期和时间值*time*中的小时值，范围是0~23。

```

HOUR('12:31:58')                     → 12
HOUR(123158)                           → 12

```

#### • MINUTE(*time*)

返回值：日期和时间值*time*中的分钟值，范围是0~59。

```

MINUTE('12:31:58')                   → 31
MINUTE(123158)                         → 31

```

#### • MONTH(*date*)

返回值：日期和时间值*date*中的月份值，范围是1~12。

```

MONTH('2002-12-01')                  → 12
MONTH(20021201)                       → 12

```

#### • MONTHNAME(*date*)

返回值：日期和时间值*date*中的月份名称（字符串）。

```

MONTHNAME('2002-12-01')               → 'December'
MONTHNAME(20021201)                   → 'December'

```

#### • NOW()

返回值：当前日期和时间。根据该函数被调用时的上下文，返回值或者是一个'CCYY-MM-DD hh:mm:ss'格式的字符串，或者是一个CCYYMMDDhhmmss格式的数值。

```
NOW() → '2002-05-14 10:19:20'
NOW() + 0 → 20020514101920
```

• **PERIOD\_ADD(*period*, *n*)**

返回值：时间段起点*period*加上*n*个月后得到的结果。返回值的格式是CCYYMM。输入参数*period*的格式可以是CCYYMM或YYMM。需要特别注意的是，这两种格式都不是MySQL惯用的日期格式。

```
PERIOD_ADD(200202, 12) → 200302
PERIOD_ADD(0202, -3) → 200111
```

• **PERIOD\_DIFF(*period1*, *period2*)**

返回值：时间段起点*period1*和*period2*做减法得到的差，即它们之间相隔的月份数。这两个输入参数的格式是CCYYMM或YYMM。需要特别注意的是，这两种格式都不是MySQL惯用的日期格式。

```
PERIOD_DIFF(200302, 200202) → 12
PERIOD_DIFF(200111, 0202) → -3
```

• **QUARTER(*date*)**

返回值：日期和时间值*date*确定的日子所在的季度值，范围是1~4。

```
QUARTER('2002-12-01') → 4
QUARTER('2003-01-01') → 1
```

• **SECOND(*time*)**

返回值：日期和时间值*time*中的秒值，范围是0~59。

```
SECOND('12:31:58') → 58
SECOND(123158) → 58
```

• **SEC\_TO\_TIME(*seconds*)**

返回值：把秒数*second*转换为相应的时间值。根据该函数被调用时的上下文，返回值或者是一个'hh:mm:ss'格式的字符串，或者是一个hhmmss格式的数值。

```
SEC_TO_TIME(29834) → '08:17:14'
SEC_TO_TIME(29834) + 0 → 81714
```

• **SUBDATE(*date*, INTERVAL *expr interval*)**

本函数是DATE\_SUB()函数的同义词。

• **SYSDATE()**

本函数是NOW()函数的同义词。

• **TIME\_FORMAT(*time*, *format*)**

返回值：按格式字符串*format*对日期和时间值*time*进行格式编排后得到的字符串。这个函数也接受DATETIME或TIMESTAMP类型的输入参数。它的格式字符串与DATEFORMAT()

函数中使用的相类似，只是只允许使用与时间有关的格式标识符而已——其他格式标识符将使它的返回值是NULL或0。

```
TIME_FORMAT('12:31:58', '%H %i')      → '12 31'
TIME_FORMAT(123158, '%H %i')          → '12 31'
```

#### • TIME\_TO\_SEC(*time*)

返回值：转换为与时间值*time*相应的秒数。这个秒数可以用SEC\_TO\_TIME()函数再次转换回原来的时间值。

```
TIME_TO_SEC('08:17:14')              → 29834
SEC_TO_TIME(29834)                   → '08:17:14'
```

如果输入参数是一个DATETIME或TIMESTAMP值，TIME\_TO\_SEC()将忽略其中的日期部分。

```
TIME_TO_SEC('2002-03-26 08:17:14')    → 29834
```

#### • TO\_DAYS(*date*)

返回值：把日期值*date*转换为从公元0年1月1日开始计算的天数，它是一个数值。这个天数可以用FROM\_DAYS()函数再次转换回原来的日期值。

```
TO_DAYS('2002-12-01')                → 731550
FROM_DAYS(731550 - 365)                → '2001-12-01'
```

如果输入参数是一个DATETIME或TIMESTAMP值，TO\_DAYS()将忽略其中的时间部分。

```
TO_DAYS('2002-12-01 12:14:37')        → 731550
```

TO\_DAYS()函数只能对格里高里历法日期（始于公元1582年）进行转换。

#### • UNIX\_TIMESTAMP()

UNIX\_TIMESTAMP(*date*)

返回值：如果不带输入参数，将返回自UNIX纪元（格林威治时间'1970-01-01 00:00:00'）开始计算的秒数；如果带一个日期值*date*作为输入参数，将返回自UNIX纪元到该日期之前所经过的秒数。输入参数*date*既可以是一个DATE、DATETIME或TIMESTAMP类型的值；也可以是一个CCYYMMDD或YYMMDD格式的数值。

```
UNIX_TIMESTAMP()                     → 1021389578
UNIX_TIMESTAMP('2002-12-01')         → 1038722400
UNIX_TIMESTAMP(20021201)              → 1038722400
```

#### • WEEK(*date*)

WEEK(*date*, *first\_day*)

返回值：如果只带一个输入参数，就将返回一个0~53之间的数值以表示日期和时间值*date*落在了一年中的第几个星期里，且每个星期的第一天从星期日开始；当带有两个输入参数时，函数WEEK()仍将返回同样的值，但每个星期的第一天到底从星期几开始将由*first\_day*参数来决定：如果*first\_day*等于0，每个星期的第一天就从星期日开始；如果



*first\_day*等于1, 每个星期的第一天就从星期一开始。

```
WEEK('2002-12-08')           → 50
WEEK('2002-12-08',0)        → 50
WEEK('2002-12-08',1)        → 49
```

如果WEEK()函数的返回值是0, 则表示给定的*date*值出现在该年度的第一个星期日或该年度的第一个星期一(假如第二个输入参数*first\_day*等于1的话)之前。

```
WEEK('2005-01-01')           → 0
DAYNAME('2005-01-01')        → 'Saturday'
WEEK('2006-01-01',1)         → 0
DAYNAME('2006-01-01')        → 'Sunday'
```

带两个输入参数的WEEK()函数最早出现于MySQL 3.22.1。

#### • WEEKDAY(*date*)

返回值: 日期和时间值*date*是星期几, 返回值是一个数值。星期几的范围是0~6, 其中0代表Monday(星期一)、6代表Sunday(星期日)。另请参见关于DAYOFWEEK()函数的介绍并注意它们在返回值方面的区别。

```
WEEKDAY('2002-12-08')        → 6
DAYNAME('2002-12-08')        → 'Sunday'
WEEKDAY('2002-12-16')        → 0
DAYNAME('2002-12-16')        → 'Monday'
```

#### • YEAR(*date*)

返回值: 日期和时间值*date*中的年份值, 范围是1 000~9 999。

```
YEAR('2002-12-01')           → 2002
YEAR(20021201)                → 2002
```

#### • YEARWEEK(*date*)

YEARWEEK(*date*, *first\_day*)

返回值: 如果只带一个输入参数, 将以CCYYWW的格式返回一个数值以表示给定日期和时间值*date*落在该年度的第几个星期里。这个返回值的取值范围是0~53, 且以星期日为每个星期的第一天。如果带有两个输入参数, YEARWEEK()函数仍将返回同样的值, 但每个星期的第一天到底是星期几将由*first\_day*参数来决定: 如果*first\_day*等于0, 每个星期的第一天就将是星期日; 如果*first\_day*等于1, 每个星期的第一天就将是星期一。

```
YEARWEEK('2006-01-01')       → 200601
YEARWEEK('2006-01-01',0)     → 200601
YEARWEEK('2006-01-01',1)     → 200552
```

有一种情况需要大家特别注意, 那就是返回值里的年份部分可能与*date*值中的年份部分不一致。这种情况发生在WEEK()函数的返回值是0(即给定的*date*值出现在该年度的第一个星期日或星期一之前)的时候。但因为YEARWEEK()函数不允许返回值中的星期部分等于0, 所以在遇到这种情况的时候, YEARWEEK()函数将以上一年的最后一个星期作为自己的返回值。如下所示:

```
WEEK('2005-01-01')           → 0
YEARWEEK('2005-01-01')       → 200452
```

YEARWEEK()函数最早出现于MySQL 3.23.8。

### C.2.6 统计类函数

统计类函数又叫做汇总类函数。它们的返回值是根据一组数据而计算出来的统计结果。这个统计结果是根据查询结果中的非NULL值统计出来的（只有COUNT(\*)函数是个例外，它将统计所有的数据行）。统计类函数既可以对整个查询结果集进行统计，也可以对查询结果按某种规则进行归组（比如查询语句里使用了GROUP BY子句的情况）后得到的各个子集进行统计。有关这方面的详细讨论见第1.4.8节中的“生成统计信息”小节。

这一小节里的示例要用到一个下面这样的mytbl数据表：它有一个整数数据列mycol，各数据行在这个数据列的值依次是1、3、5、5、7、9、9和NULL，如下所示：

```
mysql> SELECT mycol FROM mytbl;
+-----+
| mycol |
+-----+
|      1 |
|      3 |
|      5 |
|      5 |
|      7 |
|      9 |
|      9 |
|  NULL |
+-----+
```

#### • AVG(expr)

返回值：表达式expr计算结果的平均值，涉及查询结果集里的全体非NULL值。

```
SELECT AVG(mycol) FROM mytbl           → 5.5714
SELECT AVG(mycol)*2 FROM mytbl         → 11.1429
SELECT AVG(mycol*2) FROM mytbl         → 11.1429
```

#### • BIT\_AND(expr)

返回值：表达式expr计算结果的二进制与操作结果，涉及查询结果集里的全体非NULL值。

```
SELECT BIT_AND(mycol) FROM mytbl       → 1
```

#### • BIT\_OR(expr)

返回值：表达式expr计算结果的二进制或操作结果，涉及查询结果集里的全体非NULL值。

```
SELECT BIT_OR(mycol) FROM mytbl        → 15
```

#### • COUNT(expr)

COUNT(\*)

COUNT(DISTINCT expr1, expr2, ...)

返回值：COUNT(*expr*)将返回查询结果集里的非NULL值的个数。COUNT(\*)将返回查询结果集里全体数据行（不管它们是不是NULL值）的个数，如下所示：

```
SELECT COUNT(mycol) FROM mytbl           → 7
SELECT COUNT(*) FROM mytbl               → 8
```

对于ISAM和MyISAM数据表，不带WHERE子句的COUNT(\*)被优化成快速返回FROM子句所指定的数据表里的数据行总数；如果FROM子句指定了多个数据表，COUNT(\*)将返回各数据表的数据行总数的乘积，如下所示：

```
SELECT COUNT(*) FROM mytbl AS m1, mytbl AS m2 → 64
```

在MySQL 3.23.2及以后的版本里，还可以用COUNT(DISTINCT)统计出查询结果里有多少种各不相同的非NULL值，如下所示：

```
SELECT COUNT(DISTINCT mycol) FROM mytbl      → 5
SELECT COUNT(DISTINCT MOD(mycol,3)) FROM mytbl → 3
```

如果给出了多个表达式，COUNT(DISTINCT)的返回值将是全体表达式非NULL值计算结果的各不相同的组合的总数。

#### • MAX(*expr*)

返回值：表达式*expr*计算结果的最大值，涉及查询结果集里的全体非NULL值。如果用在字符串或时间值上，MAX()将返回字符串比较操作或日期和时间值比较操作中的最大值。

```
SELECT MAX(mycol) FROM mytbl → 9
```

#### • MIN(*expr*)

返回值：表达式*expr*计算结果的最小值，涉及查询结果集里的全体非NULL值。如果用在字符串或时间值上，MIN()将返回字符串比较操作或日期和时间值比较操作中的最小值。

```
SELECT MIN(mycol) FROM mytbl → 1
```

#### • STD(*expr*)

返回值：表达式*expr*计算结果的标准方差，涉及查询结果集里的全体非NULL值。

```
SELECT STD(mycol) FROM mytbl → 2.7701
```

#### • STDDEV(*expr*)

本函数是STD()函数的同义词。

#### • SUM(*expr*)

返回值：表达式*expr*计算结果的总和，涉及查询结果集里的全体非NULL值。

```
SELECT SUM(mycol) FROM mytbl → 39
```

### C.2.7 数据加密类函数

这些函数用来完成各种与数据安全有关的操作，如字符串的加密或解密。在这类函数中，有一些是成对出现的，其中一个用来进行加密，另一个则用来进行解密。这些成对出现的函数通常都要使用同一个字符串来充当密钥或口令——要想在解密后得到原先的数据，就必须使用

在加密它们时使用的同一个密钥来进行解密；否则，解密结果即使不是一堆毫无价值的乱码，也不会是原来的数据。

加密函数的返回值通常都是一些二进制字符串（即需要区分字母的大小写情况）。因此，如果你打算把它们保存到数据库里，就应该使用BLOB类型的数据列。

- AES\_DECRYPT(*str*, *key\_str*)

返回值：采用AES算法以密钥*key\_str*对字符串*str*进行解密而得到的字符串。*str*应该是一个由AES\_ENCRYPT()函数使用同一个密钥*key\_str*进行加密而得到的二进制字符串。如果输入参数中有NULL值，则返回NULL。

```
AES_DECRYPT(AES_ENCRYPT('secret', 'scramble'), 'scramble')
→ 'secret'
```

AES\_DECRYPT()函数最早出现于MySQL 4.0.2。

- AES\_ENCRYPT(*str*, *key\_str*)

返回值：采用AES算法以密钥*key\_str*对字符串*str*进行加密而得到的二进制（即需要区分字母的大小写情况）字符串。AES是Advanced Encryption Standard（高级加密标准）的缩写，它使用的密钥长度是128位。如果输入参数中有NULL值，则返回NULL。本函数的加密结果可以用AES\_DECRYPT()函数和同一个密钥解密为原先的字符串。

AES\_ENCRYPT()函数最早出现于MySQL 4.0.2。

- DECODE(*str*, *key\_str*)

返回值：用密钥*key\_str*对字符串*str*进行解密而得到的结果。*str*应该是一个由ENCODE()函数用同一个密钥*key\_str*加密而得到的字符串。如果字符串*str*是NULL值，则返回NULL。

```
DECODE(ENCODE('secret', 'scramble'), 'scramble') → 'secret'
```

- DES\_DECRYPT(*str*)

DES\_DECRYPT(*str*, *key\_str*)

返回值：采用DES算法对字符串*str*进行解密而得到的结果。*str*应该是一个用DES\_ENCRYPT()函数进行加密而得到的二进制字符串。如果系统上的SSL支持功能没有被激活或者解密操作失败，DES\_DECRYPT()将返回NULL。

如果给出了*key\_str*参数，DES\_DECRYPT()函数就将把它用做解密密钥。如果没有给出*key\_str*参数，DES\_DECRYPT()函数就将使用一个来自服务器DES密钥文件里的密钥来对字符串*str*进行解密；这个密钥的编号将由字符串*str*第一个字节里的0~6位决定，而密钥文件在服务器里的存放地点则是由你（或别人）在启动服务器时用--des-key-file选项指定的。如果加密和解密时使用的密钥不一致，就会导致毫无意义的结果。

如果字符串*str*看起来不像是一个经加密而得到的结果，比如字符串*str*第一个字节的第7位没有被置位（即等于0）时，DES\_DECRYPT()函数将不加改变地原样返回字符串*str*。

只有拥有SUPER权限的用户才能使用只带一个参数的DES\_DECRYPT()函数。

DES\_DECRYPT()函数最早出现于MySQL 4.0.1。

- DES\_ENCRYPT(*str*)

DES\_ENCRYPT(*str*, *key\_num*)

**DES\_ENCRYPT(*str*, *key\_str*)**

返回值：采用DES算法对字符串*str*进行加密而得到的二进制字符串。本函数的加密结果可以用DES\_DECRYPT()函数解密为原先的字符串。如果系统上的SSL支持功能没有被激活或者加密操作失败，DES\_ENCRYPT()将返回NULL。

如果给出了*key\_str*参数，DES\_ENCRYPT()函数就将把它用做加密密钥。如果给出了*key\_num*参数（它应该是一个0~9之间的整数），DES\_DECRYPT()函数就将使用服务器DES密钥文件里编号为*key\_num*的密钥来对字符串*str*进行加密。如果*key\_str*和*key\_num*参数都没有给出，就将使用DES密钥文件里的第一个密钥（注意：它与你把*key\_num*参数设置为0时使用的密钥不一定是同一个）来进行加密。

加密结果字符串的第一个字节能让我们知道加密工作是如何进行的。这个字节的第7位应该被置位为1，而第0~6位则构成了一个密钥编号：如果这个编号是一个0~9之间的数字，就说明加密工作是用服务器DES密钥文件里对应于该编号的密钥来完成的。如果这个编号等于127，就说明加密工作是用一个*key\_str*参数来完成的。比如说，如果加密时使用的密钥编号为3，加密结果字符串第一个字节的值就应该是131（即128+3）。如果加密时使用的是一个*key\_str*参数，加密结果字符串第一个字节的值就应该是255（即128+127）。

对于使用*key\_num*参数的加密操作，服务器将从某个DES密钥文件里读出相应的密钥字符串，而这个DES密钥文件在服务器里的存放地点则是由你（或别人）在启动服务器时用--des-key-file选项指定的。密钥在密钥文件里的存放格式如下所示：

```
key_num key_str
```

其中，*key\_num*是一个0~9之间的整数，而*key\_st*则是与之对应的加密密钥；*key\_num*与*key\_st*之间至少要用一个空白符加以分隔。密钥文件里的各行允许按任意先后顺序出现。

DES\_ENCRYPT()函数并不要求用户必须拥有SUPER权限才能使用来自DES密钥文件里的密钥，这与DES\_DECRYPT()函数是不同的。（谁都可以使用DES密钥文件里的密钥去加密自己的信息，但只有那些有足够权限的用户才能去解密出信息。）

DES\_ENCRYPT()函数最早出现于MySQL 4.0.1。

- **ENCODE(*str*, *key\_str*)**

返回值：用密钥*key\_str*对字符串*str*进行加密而得到的二进制字符串。本函数的加密结果可以用DECODE()函数和同一个密钥解密为原先的字符串。

- **ENCRYPT(*str*)**

**ENCRYPT(*str*, *salt*)**

返回值：字符串*str*的加密结果；如果输入参数中有NULL值，则返回NULL。这是一个不可逆的加密过程。*salt*参数（如果给出的话）必须是一个由两个字符组成的字符串（在MySQL 3.22.16及以后的版本里，*salt*的长度允许超过两个字符）。对于一个给定的*salt*值，不管你对字符串*str*进行多少次加密，其结果都将是一样的。如果不带*salt*参数，ENCRYPT()函数对字符串*str*每次加密的结果都将是不同的。

ENCRYPT('secret', 'AB')	→ 'ABS5SGh1EL6bk'
ENCRYPT('secret', 'AB')	→ 'ABS5SGh1EL6bk'
ENCRYPT('secret')	→ '9u0hlzMkCx9N2'
ENCRYPT('secret')	→ 'avGJcOP2vakBE'



ENCRYPT()函数是在UNIX操作系统的crypt()系统调用的基础上实现的。因此,如果crypt()调用在你的系统上不可用,ENCRYPT()函数就将总是返回一个NULL值。又因为不同系统上的crypt()调用有细微的差异(比如说,在某些系统上,crypt()只对加密字符串的前八个字符进行加密),所以ENCRYPT()函数在不同系统上也会有不同的表现。

- MD5(*str*)

返回值:用MD5信息标记算法为字符串*str*生成的一个128位的校验和。MD5信息汇编算法是由RSA Data Security, Inc. (RSA数据安全公司)研发的。这个返回值是一个由32个十六进制数字构成的字符串;但如果字符串*str*是NULL,那么返回值也将是NULL。

```
MD5('secret') → '5ebe2294ecd0e0f08eab7690d2a6ee69'
```

MD5()函数最早出现于MySQL 3.23.2。另请参阅关于SHA1()函数的介绍。

- PASSWORD(*str*)

返回值:根据字符串*str*而计算出来的密文字符串,其格式与MySQL在它存放用户口令的权限表里使用的格式完全一样。这是一个不可逆的加密过程。

```
PASSWORD('secret') → '428567f408994404'
```

需要提醒大家注意的是,PASSWORD()函数所使用的算法与UNIX用来加密UNIX账户口令的算法是不一样的——如果你想要的是后一种加密效果,就应该使用ENCRYPT()函数。

- SHA(*str*)

本函数是SHA1()函数的同义词。

- SHA1(*str*)

返回值:用SHA (Secure Hash Algorithm, 安全散列算法) 为字符串*str*生成的一个160位的校验和。这个返回值是一个由40个十六进制数字构成的字符串;但如果字符串*str*是NULL,那么返回值也将是NULL。

```
SHA1('secret') → 'e5e9fa1ba31ecd1ae84f75caaa474f3a663f05f4'
```

SHA1()函数最早出现于MySQL 4.0.2。另请参阅关于MD5()函数的介绍。

## C.2.8 其他函数

那些无法划归到以上各大类里去的函数都收录在本小节里。

- BENCHMARK(*n*, *expr*)

对表达式*expr*反复求值*n*次。BENCHMARK()是一个不同寻常的函数:首先,它只能在mysql客户程序里使用;其次,它的返回值永远是0,没有多大的实际价值。人们真正感兴趣的是mysql程序显示在查询结果下面那一行里的时间值:

```
mysql> SELECT BENCHMARK(1000000,PASSWORD('secret'));
+-----+
| BENCHMARK(1000000,PASSWORD('secret')) |
+-----+
| 0 |
+-----+
1 row in set (2.35 sec)
```

不过，这个值只是客户（程序）在等待查询结果从服务器返回时逝去的时间，而不是服务器上的CPU时间，所以它只能大概地表明服务器对表达式`expr`的求值有多快。这个时间会受到很多因素的影响，比如服务器的负载情况、服务器在查询到达时是正处于运行状态还是刚被切换出系统内存等等。总之，要多进行几次查询才能得出一个有代表性的数值来。

`BENCHMARK()`函数最早出现于MySQL 3.22.15。

- `BIT_COUNT(n)`

返回值：输入参数`n`被置位（即等于1）的位数，`n`将被当做是一个BIGINT（64位整数）值。如果`n`是NULL，那么返回值也将是NULL。

<code>BIT_COUNT(0)</code>	→ 0
<code>BIT_COUNT(1)</code>	→ 1
<code>BIT_COUNT(2)</code>	→ 1
<code>BIT_COUNT(7)</code>	→ 3
<code>BIT_COUNT(-1)</code>	→ 64
<code>BIT_COUNT(NULL)</code>	→ NULL

- `BIT_LENGTH(str)`

返回值：字符串`str`以位计算的长度；如果`str`是NULL，那么返回值也将是NULL。

<code>BIT_LENGTH('abc')</code>	→ 24
<code>BIT_LENGTH('a long string')</code>	→ 104

`BIT_LENGTH()`函数最早出现于MySQL 4.0.2，是为与ODBC保持兼容而引入的。

- `CONNECTION_ID()`

返回值：当前连接的连接标识符，也就是MySQL服务器为当前客户（程序）所分配的线程标识符。

<code>CONNECTION_ID()</code>	→ 10146
------------------------------	---------

`CONNECTION_ID()`函数最早出现于MySQL 3.23.14。

- `DATABASE()`

返回值：一个包含着当前数据库名称的字符串；如果没有当前数据库，则返回NULL。

<code>DATABASE()</code>	→ 'sampdb'
-------------------------	------------

- `FOUND_ROWS()`

返回值：前一条SELECT语句在没有LIMIT子句的情况下应该返回的数据行的个数。比如说，下面这个查询最多只能返回10个数据行：

```
mysql> SELECT * FROM mytbl LIMIT 10;
```

要想知道这条语句在没有LIMIT子句的情况下会返回多少个数据行，就应该像下面这样做：

```
mysql> SELECT SQL_CALC_FOUND_ROWS * FROM mytbl LIMIT 10;
mysql> SELECT FOUND_ROWS();
```

`FOUND_ROWS()`函数最早出现于MySQL 4.0.0。

- `GET_LOCK(str, timeout)`

`GET_LOCK()`与`RELEASE_LOCK()`和`IS_FREE_LOCK()`的配合使用为MySQL提供了咨询

锁定机制 (advisory locking) 或者叫合作性锁定机制 (cooperative locking)。利用这几个函数, 可以编写出彼此合作的应用程序, 这类应用程序能够根据有关数据锁的状态来协调彼此的执行情况。

GET\_LOCK()函数中的字符串参数`str`是某个数据锁的名字, 而数值参数`timeout`则是以秒计算的等待时间。如果在给定的等待时间内成功地申请到了指定的数据锁, GET\_LOCK()函数将返回1; 如果因超时而没有获得数据锁, 返回0; 如果执行出错, 返回NULL。注意, `timeout`参数给出的是等待获得某个数据锁的时间, 而不是该数据锁的生效时间。在获得某个数据锁之后, 只要你没有释放它, 它的效力就将一直保持下去。

我们来看下面这个调用, 它准备申请一个名为'Nellie'的数据锁, 并愿意为此等待10秒钟:

```
GET_LOCK('Nellie',10)
```

这种数据锁只对作为它名字的那个字符串有效力。它对数据库、数据表或者数据表里的数据行或数据列都没有效力。换句话说, 它完全无法阻止其他客户 (程序) 对数据库、数据表或者数据表里的数据行或数据列做任何事情, 这也正是它得名咨询锁定机制的原因——这种锁定机制只不过是提供了一种简单的协调手段, 让一组需要相互合作的应用程序可以判断数据锁是否已经生效并据此来协调各自的执行情况。

这种施加在某个名字 (即字符串) 上的数据锁一旦为某个客户 (程序) 拥有, 就将阻塞其他客户 (程序) 试图锁定该名字的尝试。如果这是一个与服务器建立有多个连接的多线程客户程序, 那就还将阻塞其他线程试图锁定该名字的尝试。假如客户1已经锁定了字符串'Nellie'。如果现在有一个客户2想要锁定同一个字符串, 那么, 在客户1释放这把锁或者客户2自己设定的等待时间到达之前, 客户2将一直处于阻塞状态。如果是前一种情况 (客户1释放这把锁), 客户2将如愿以偿地获得这把锁; 如果是后一种情况 (客户2等待超时), 客户2的数据锁申请调用将以失败告终。

因为两个客户 (程序) 不可能同时锁定同一个给定的字符串, 所以那些同意把该字符串用做数据锁的应用程序就可以利用该数据锁的状态来协调有关的操作。比如说, 如果数据表里有一个数据行存放着每次只允许一个客户 (程序) 去修改的重要数据, 就应该考虑在有关的应用程序里为这个数据行安排一个合作性数据锁来协调有关的操作——即以团结协作为目的去锁定这个数据行。

这种数据锁的释放操作很简单, 以数据锁的名字为输入参数去调用一次RELEASE\_LOCK()函数就行了。如下所示:

```
RELEASE_LOCK('Nellie')
```

如果成功地释放了数据锁, RELEASE\_LOCK()函数将返回1; 如果数据锁是由另一个连接拥有的, 就将返回0 (也就是说, 你只能释放由你本人拥有的数据锁); 如果你打算释放的数据锁不存在, 将返回NULL。

MySQL规定每个客户 (程序) 连接同一时间只能拥有一把这样的字符串锁, 因此, 当拥有某个字符串锁的客户 (程序) 发出另一条GET\_LOCK()调用时, 它原先拥有的数据锁将自动被释放。在这种情况下, 旧数据锁将在获得新数据锁之前被释放——哪怕新、旧数据

锁的名字完全相同也将如此。此外，当客户（程序）与服务器之间的连接被断开时，客户（程序）拥有的数据锁也将被释放。因此，如果你有一个需要运行很长时间的客户（程序），就必须警惕下面这种情况的发生：客户（程序）因为长时间没有操作动作而导致服务等待超时（即被服务器认为你已经离开却忘了断开连接），而当服务器断开与客户（程序）的连接时，它拥有的数据锁就将被自动释放。

如果你只是想知道字符串`str`上的数据锁是否处于生效状态而不是想花时间去获得它，可以简单地用一个`GET_LOCK(str, 0)`调用来达到这一目的。不过，如果这把数据锁恰好处于空闲状态，你就会有“无意”中锁定这个字符串，因此，应该立刻再调用一次`RELEASE_LOCK(str)`函数。

`IS_FREE_LOCK(str)`函数是专门用来测试字符串锁的当前状态的：如果字符串锁可用（即字符串`str`当前没有被用做一个数据锁），它将返回1；如果字符串锁正被其他客户（程序）锁定，返回0；如果执行出错，返回NULL。

如果字符串输入参数`str`是NULL值，这三个函数的返回值都将是NULL。

#### • `INET_ATON(str)`

返回值：把四字段IP地址字符串`str`转换为一个整数。如果`str`不是一个合法的IP地址，则返回NULL。

<code>INET_ATON('64.28.67.70')</code>	→ 1075594054
<code>INET_ATON('255.255.255.255')</code>	→ 4294967295
<code>INET_ATON('256.255.255.255')</code>	→ NULL
<code>INET_ATON('www.mysql.com')</code>	→ NULL

`INET_ATON()`函数最早出现于MySQL 3.23.15。

#### • `INET_NTOA(n)`

返回值：把IP地址的整数表示形式`n`转换为四字段IP地址字符串。如果`n`不能代表一个合法的IP地址，则返回NULL。

<code>INET_NTOA(1075594054)</code>	→ '64.28.67.70'
<code>INET_NTOA(2130706433)</code>	→ '127.0.0.1'

`INET_NTOA()`函数最早出现于MySQL 3.23.15。

#### • `IS_FREE_LOCK(str)`

返回值：名为`str`的合作性数据锁的当前状态。这个函数要与`GET_LOCK()`和`RELEASE_LOCK()`函数配合使用。参见`GET_LOCK()`条目中的详细解释。

`IS_FREE_LOCK()`函数最早出现于MySQL 4.0.2。

#### • `LAST_INSERT_ID()`

`LAST_INSERT_ID(expr)`

返回值：如果不带参数，将返回本次服务器会话期间最近一次生成的`AUTO_INCREMENT`值；如果尚未生成过这样的编号，则返回0。带`expr`参数的`LAST_INSERT_ID()`函数通常都出现在`UPDATE`语句里，它的效果是使表达式`expr`的值被当做是最近一次生成的`AUTO_INCREMENT`值，可以利用这一特性来生成各种序列编号。

对本函数的详细介绍可以参见第2章。不管调用的是`LAST_INSERT_ID()`函数的哪一种形

式，它们的返回值都将由MySQL服务器按“谁生成，谁拥有”的原则为每个连接分别保存，即使其他客户（程序）又生成了新的自动编号，也不会影响或改变所生成的编号值。带`expr`参数的`LAST_INSERT_ID()`函数最早出现于MySQL 3.22.9。

- `LOAD_FILE(filename)`

返回值：读取文件`filename`并把它的内容返回为一个字符串。这个文件必须是存放在服务器上的，必须以绝对（完整）路径名的形式指定，必须是一个全局可读的文件（以确保你不是在试图读取一个受保护的文件）。因为这个文件必须是存放在服务器上的，所以你还必须拥有FILE权限。只要以上条件有不满足的，`LOAD_FILE()`函数就将返回NULL。

`LOAD_FILE()`函数最早出现于MySQL 3.23.0。

- `MASTER_POS_WAIT(log_file, pos)`

这个函数主要用来测试主镜像服务器。它将阻塞主服务器直到从服务器到达日志文件中的给定位置为止。如果从服务器已经到达给定位置，这个函数将立刻返回。如果从服务器根本就没有运行，主服务器将一直阻塞到从服务器开机启动并到达给定位置为止。

`MASTER_POS_WAIT()`函数的返回值是主服务器在从服务器到达给定位置之前还必须等待的日志文件事件的个数。如果执行出错或者主服务器信息尚未被初始化，则返回NULL。

`MASTER_POS_WAIT()`函数最早出现于MySQL 3.23.32。

- `RELEASE_LOCK(str)`

释放名为`str`的合作性数据锁。这个函数必须与`GET_LOCK()`函数配合使用。参见`GET_LOCK()`条目中的详细解释。

- `SESSION_USER()`

本函数是`USER()`函数的同义词。

- `SYSTEM_USER()`

本函数是`USER()`函数的同义词。

- `USER()`

返回值：被表示为'`user@host`'格式的字符串的当前客户（程序）的使用者，其中的`user`是使用者的用户名，`host`是用来建立当前客户连接的主机的名字。

<code>USER()</code>	→ ' <code>sampadm@localhost</code> '
<code>SUBSTRING_INDEX(USER(), '@', 1)</code>	→ ' <code>sampadm</code> '
<code>SUBSTRING_INDEX(USER(), '@', -1)</code>	→ ' <code>localhost</code> '

在MySQL 3.22.1之前的版本里，`USER()`函数的返回值只包含现在的用户名部分。

- `VERSION()`

返回值：一个描述服务器版本情况的字符串。

<code>VERSION()</code>	→ ' <code>4.0.3-beta-log</code> '
------------------------	-----------------------------------

在这个函数的返回值里，服务器版本号的后面可能还跟有一个或者多个后缀。下面是一些比较常见的后缀及其含义：



- -alpha、-beta或-gamma——表明该MySQL服务器版本的稳定性。
- -debug——表示该服务器运行在调试模式下。
- -demo——表示该服务器运行在演示模式下（只用在MySQL 3.23.30或更早的版本里）。
- -embedded——表示是一个嵌入式MySQL服务器libmysqld。
- -log——表示日志已激活。
- -max——表示该服务器里编译有一些附加功能。
- -nt——表示该服务器是为基于Windows NT的系统而建立的。



## 附录D SQL语法指南

本附录的主要内容包括：

- MySQL所支持的SQL语句的使用方法。
- 如何在SQL语句里设置和使用用户定义的变量。
- 在SQL代码里编写注释的语法。注释的主要用途有两个，一是写出不会被MySQL服务器执行的描述性文字；二是隐藏MySQL特有的关键字（这些关键字会被MySQL执行，但其他数据库服务器会忽略它们）。

MySQL的开发和完善工作一直没有停过，所以它能够支持的SQL语句也一直在丰富之中。如果你想知道MySQL最近又增加了哪些新功能，请查阅MySQL官方网站（它的URL地址是<http://www.mysql.com/>）上的在线MySQL Reference Manual（MySQL参考手册）。

在介绍SQL语法的时候，本附录使用以下约定：

- 可选信息将放在一对方括号（`[]`）里。
- 垂直线字符（`|`）用来分隔参数清单里的多选一替换项。若参数清单出现在一对方括号里，则表示可以选取一个替换项；如果参数清单出现在一对花括号（`{ }`）里，则表示必须选取一个替换项。
- 省略号（`...`）表示该省略号前面的那个项目可以重复多次地出现。
- `n`代表一个整数。
- `'string'`代表一个字符串值。单引号值`'file_name'`或`'pattern'`表示更特定类型的值，比如一个文件名或者一个匹配模式。

如果未做特别说明，本附录列出的SQL语句至少从MySQL 3.22.0版本起就已经出现在MySQL里了。

### D.1 SQL语句

本节将对MySQL支持的SQL语句的语法和含义进行描述。给定一条SQL语句，如果你没有足够的权限去执行它，这条语句的执行就会失败。比如说，如果你没有访问`db_name`数据库的权限，所发出的`USE db_name`语句就不会执行成功。

#### D.1.1 ALTER DATABASE

语法：`ALTER DATABASE db_name action_list`

这条语句用来改变数据库的全局特性。`action_list`给出了一个或者多个动作，这些动作要用逗号分隔开。但目前只有一个可用的动作：

`[ DEFAULT ] CHARACTER SET charset`

*charset*可以是某个字符集的名字；也可以是关键字DEFAULT，表示数据库将使用服务器的当前字符集作为其默认字符集。

这条语句要求你必须具有数据库上的ALTER权限。

这条语句最早出现于MySQL 4.1版本。

### D.1.2 ALTER TABLE

语法：ALTER [ IGNORE ] TABLE *tbl\_name* *action\_list*

ALTER TABLE允许你重新命名一个数据表或者更改它的结构。在使用这条命令的时候，需要给出一个数据表名*tbl\_name*以及一个或者多个将对数据表进行的操作动作。如果某个操作动作会使新数据表里的惟一化索引出现重复的键值，就需要选用IGNORE关键字：如果没有IGNORE关键字，ALTER TABLE语句的动作效果将被撤销；如果有这个关键字，会导致惟一化索引出现重复键值的那些数据行将被删除掉。

除重新命名数据表这一个操作动作外，ALTER TABLE语句的其他动作将先根据原始数据表创建一个新数据表，然后再对新创建的数据表依次实施各个动作。如果执行出错，新数据表将被丢弃而原始数据表则保持不变。如果全部操作动作都执行成功，新数据表就将取代原始数据表（原始数据表将被丢弃）。在这一过程中，其他客户（程序）依然可以从原始数据表里读取数据，但试图对之进行写操作的客户（程序）将被阻塞直到ALTER TABLE语句执行完毕——被阻塞的写操作将实施在新数据表上。

*action\_list*负责给出ALTER TABLE语句将要执行的操作动作，多个动作之间要用逗号隔开。下面是ALTER TABLE语句目前所支持的各种操作动作及其含义：

- ADD [ COLUMN ] *col\_declaration* [ FIRST | AFTER *col\_name* ]

给数据表增加一个数据列。*col\_declaration*是新增数据列的定义，它与CREATE TABLE语句中的数据列定义格式完全相同。如果给出了FIRST关键字，新增数据列将成为该数据表的第一个数据列；如果给出了AFTER *col\_name*，新增数据列将被安排在数据列*col\_name*的后面；如果没有替新增数据列安排位置，它将成为该数据表的最后一个数据列。

```
ALTER TABLE t ADD id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY;
ALTER TABLE t ADD id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY FIRST;
ALTER TABLE t ADD id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY
    AFTER suffix;
```

- ADD [ COLUMN ] ( *create\_definition*, ... )

给数据表增加一个或者多个数据列或索引。每个*create\_definition*都是一个数据列或索引定义，它们之间以逗号分隔，格式则与CREATE TABLE语句中的数据列或索引的定义格式完全相同。这个语法最早出现于MySQL 3.23.11版本。

- ADD [ CONSTRAINT *name* ]

FOREIGN KEY [ *index\_name* ] ( *index\_columns* ) *reference\_definition*

给数据表增加一个外键定义。这个操作动作只能用在InnoDB数据表上。*index\_columns*是一个或者多个彼此以逗号分隔的数据列名字，新增加的外键就将建立在这些数据列上。

任何CONSTRAINT或*index\_name*将被忽略 (MySQL现在还没有实现相应的动作)。 *reference\_definition*定义了外键与父数据表的关联关系, 详细情况请参见CREATE TABLE条目中的有关内容。ADD FOREIGN KEY语法最早出现于MySQL 3.23.50版本。(虽然还存在着一个相应的DROP FOREIGN KEY子句, 但MySQL现在还没有实现相应的动作, 遇到它时会忽略过去。)

```
ALTER TABLE child
    ADD FOREIGN KEY (par_id) REFERENCES parent (par_id) ON DELETE CASCADE;
```

- ADD FULLTEXT [ KEY | INDEX ] [ *index\_name* ] ( *index\_columns* )

给MyISAM数据表增加一个FULLTEXT索引。*index\_columns*是一个或者多个彼此以逗号分隔的数据列名字, 新增加的FULLTEXT索引就将建立在这些数据列上。ADD FULLTEXT语法最早出现于MySQL 3.23.23版本。

```
ALTER TABLE poetry ADD FULLTEXT (author,title,stanza);
```

- ADD INDEX [ *index\_name* ] ( *index\_columns* )

给数据表增加一个索引。*index\_columns*是一个或者多个彼此以逗号分隔的数据列名字, 新增加的索引就建立在这些数据列上。对于CHAR或VARCHAR数据列, 可以对数据列的前缀进行索引, 即通过*col\_name*(*n*)语法来指定只对数据列值的前*n*个字节进行索引。对于BLOB或TEXT数据列, 必须给出一个前缀值; MySQL不允许对整个数据列进行索引。*index\_name*是这个索引的名字, 如果没有给出这个名字, MySQL就将自动使用第一个被索引数据列的名字来作为这个索引的名字。

- ADD PRIMARY KEY ( *index\_columns* )

在给定数据列上增加一个主键 (primary key)。这个主键的名字是PRIMARY。*index\_columns*参数的用途和用法与它在ADD INDEX子句里的情况相同。如果主键已经存在或者如果任何一个有关的数据列里允许使用NULL值, 这个操作动作将报告出错。

```
ALTER TABLE president ADD PRIMARY KEY (last_name, first_name);
```

- ADD UNIQUE [ *index\_name* ] ( *index\_columns* )

给指定数据表*tbl\_name*增加一个惟一化索引 (即各索引项的取值彼此不同)。*index\_name*和*index\_columns*的含义及用法与ADD INDEX子句中的情况相同。

```
ALTER TABLE absence ADD UNIQUE id_date (student_id, date);
```

- ALTER [ COLUMN ] *col\_name* { SET DEFAULT *value* | DROP DEFAULT }

改变指定数据列的默认值。这个子句既可以用来设定一个新的默认值, 也可以丢弃当前的默认值。在后一种情况里, 新默认值将按照CREATE TABLE语句条目中的描述进行设定。

```
ALTER TABLE event ALTER type SET DEFAULT 'Q';
ALTER TABLE event ALTER type DROP DEFAULT;
```

- CHANGE [ COLUMN ] *col\_name* *col\_declaration* [ FIRST | AFTER *col\_name* ]

改变指定数据列的名称和定义。*col\_name*是数据列现在的名字; *col\_declaration*则是数据列的新定义, 其格式与CREATE TABLE语句所使用的格式相同。注意: 必须在*col\_declaration*

里给这个数据列起一个新名字，如果不想改变它的名字，就必须把它现在的名字写两遍。关键字FIRST或AFTER最早出现于MySQL 4.0.1版本，它们的用途和用法与ADD COLUMN子句中的情况相同。

```
ALTER TABLE student CHANGE name name VARCHAR(40);
ALTER TABLE student CHANGE name student_name CHAR(30) NOT NULL;
```

#### • DISABLE KEYS

在默认的情况下，当你对MyISAM数据表做出修改时，它的各种非惟一化索引都会被自动更新。DISABLE KEYS子句的作用就是禁用这种索引自动更新机制（ENABLE KEYS子句将重新激活这种索引自动更新机制）。DISABLE KEYS最早出现于MySQL 4.0版本。

```
ALTER TABLE score DISABLE KEYS;
```

#### • DROP [ COLUMN ] *col\_name* [ RESTRICT | CASCADE ]

从数据表里丢弃（删除）指定的数据列。如果该数据列还是某个索引的组成部分，它将同时从那个索引里被删除。如果组成某个索引的数据列全都被删除了，该索引也将被删除。

```
ALTER TABLE president DROP suffix;
```

关键字RESTRICT或CASCADE不起作用。它们是为了使MySQL与其他数据库保持兼容而添加的，但在MySQL数据库系统里会被忽略。

#### • DROP INDEX *index\_name*

从数据表里删除指定的索引。

```
ALTER TABLE member DROP INDEX name;
```

#### • DROP PRIMARY KEY

删除数据表的主键。若数据表里没有被创建为PRIMARY KEY的惟一化索引，但有一个或者多个UNIQUE索引，则将删除它们当中的第一个。

```
ALTER TABLE president DROP PRIMARY KEY;
```

#### • ENABLE KEYS

对于MyISAM数据表，重新激活被DISABLE KEYS子句禁用的非惟一化索引自动更新机制。ENABLE KEYS最早出现于MySQL 4.0版本。

```
ALTER TABLE score ENABLE KEYS;
```

#### • MODIFY [ COLUMN ] *col\_declaration* [ FIRST | AFTER *col\_name* ]

改变数据列的定义。数据列定义*col\_declaration*的格式与CREATE TABLE语句所使用的格式相同。注意：*col\_declaration*必须以某个现有的数据列的名字开始，即改变该数据列的定义。MODIFY最早出现于MySQL 3.22.16版本；关键字FIRST或AFTER最早出现于MySQL 4.0.1版本，它们的用途和用法与ADD COLUMN子句中的情况相同。

```
ALTER TABLE student MODIFY name VARCHAR(40) DEFAULT '' NOT NULL;
```

#### • ORDER BY *col\_list*

*col\_list*是一个或者多个彼此以逗号分隔的数据列名字，数据表里的数据行将根据这些数据



列进行排序。默认的排序顺序是升序。你可以在数据列名字的后面加上关键字ASC或DESC来明确地指定按数据列的升序或降序进行排序。对数据表进行排序能够提高检索速度,改善性能。不过,如果数据表在进行完ORDER BY操作之后又发生了修改,就会打乱刚排好的顺序,所以这个操作只对那些今后不会再发生修改的数据表才有意义和用处。这个选项最早出现于MySQL 3.23.28版本。

```
ALTER TABLE score ORDER BY event_id, student_id;
```

- **RENAME [ TO | AS ] *new\_tbl\_name***

把数据表更名为*new\_tbl\_name*。

```
ALTER TABLE president RENAME TO prez;
```

在MySQL 3.23.17之前的版本里,关键字TO和AS都不存在。MySQL 3.23.17版本增加了一个可选的关键字TO; MySQL 3.23.23版本又增加了一个可选的关键字AS。

- ***table\_options***

数据表选项;其用途和格式与CREATE TABLE语句中的*table\_options*部分相同,详细情况请参阅关于CREATE TABLE语句的介绍。

```
ALTER TABLE score TYPE = MYISAM CHECKSUM = 1;
```

```
ALTER TABLE sayings CHARACTER SET utf8;
```

### D.1.3 ANALYZE TABLE

语法: **ANALYZE { TABLE | TABLES } *tbl\_name* [, *tbl\_name* ] . . .**

让MySQL对数据表进行分析,把数据表各索引的键值分布情况统计并保存起来。它适用于MyISAM和BDB数据表,且要求你必须拥有各指定数据库上的SELECT和INSERT权限。在分析工作完成之后,SHOW INDEX输出报告里的Cardinality列将给出索引里有多少彼此不同的值。利用这条语句得到的分析结果,优化器就能在今后的查询里更快地完成某些特定的关联操作。

数据表的分析操作需要读锁定,在分析期间,对数据表的写操作将被阻塞。如果你已经对某个数据表进行过分析且在分析工作完成后没有修改过那个数据表,再次发出一条对该数据表进行分析的ANALYZE TABLE命令将不会再次对之进行分析。

ANALYZE TABLE语句产生的输出报告的格式与CHECK TABLE语句相同,请参阅有关条目。

ANALYZE TABLE最早出现于MySQL 3.23.14版本。

### D.1.4 BACKUP TABLE

语法: **BACKUP { TABLE | TABLES } *tbl\_name* [, *tbl\_name* ] . . . TO '*dir\_name*'**

把指定数据表拷贝到'*dir\_name*'指定的目录里去, '*dir\_name*'应该是一个以完整路径名形式给出的MySQL服务器主机上的目录,备份文件将被写入到这个目录里去。BACKUP TABLE命令只适用于MyISAM数据表,且要求你必须拥有各指定数据库上的SELECT和FILE权限。它将把数据表的定义文件和数据文件(.frm和.MYD文件)拷贝到指定的目录里去,这些文件是恢复数据表的最低要求。它不拷贝索引文件,因为索引文件可以用数据表的定义文件和数据文件(通

过RESTORE TABLE命令)重新创建出来。

在备份过程中,各数据表将依次被加上一个读锁定。如果你打算用一条BACKUP TABLE命令来备份多个数据表,就可能出现这样的情况:当你正在备份前一个数据表的时候,后面的数据表可能被修改;或者当你正在备份后一个数据表的时候,前面的数据表又被修改了。如果你想让这些数据表的备份与它们在你发出BACKUP TABLE命令时的样子完全一致,就需要在备份工作开始之前先用LOCK TABLE命令锁定它们,在备份工作完成之后再用UNLOCK TABLE命令解除对它们的锁定。当然,这种做法将使其他客户(程序)对这些数据表的写操作被阻塞更长的时间。

BACKUP TABLE命令所创建的文件将由用来运行MySQL服务器的账户拥有。已有的数据表备份文件将被覆盖。

BACKUP TABLE命令最早出现于MySQL 3.23.25版本。

下面这条语句将把对应于数据表t的t.frm和t.MYD文件备份到MySQL服务器主机上的/var/mysql/bkup目录里去:

```
BACKUP TABLE t TO '/var/mysql/bkup';
```

### D.1.5 BEGIN

**语法:** BEGIN [ WORK ]

事务处理机制中的第一条语句,它将关闭自动提交模式直到你发出下一条COMMIT或ROLLBACK语句。在自动提交模式关闭期间执行的SQL语句将作为一个整体被提交或者被回滚。

在提交或者回滚了某个事务之后,自动提交模式将被恢复为发出BEGIN语句之前的原样。如果你想以手动方式来明确地切换自动提交模式,请使用SET AUTOCOMMIT语句(参见SET语句中的有关条目)。

在某个事务尚未结束(即尚未发出COMMIT或ROLLBACK语句)之前又发出一条BEGIN语句的做法相当于隐含地提交了这个事务。

BEGIN语句最早出现于MySQL 3.23.17版本,它的同义语句BEGIN WORK最早出现于MySQL 3.23.19版本。

### D.1.6 CHANGE MASTER

**语法:** CHANGE MASTER TO master\_defs

在镜像机制中的从服务器上改变其配置参数,比如将要连接到哪一个主服务器、如何与之建立连接、使用哪些日志等等。master\_defs是一组以逗号分隔的参数定义列表,参数的定义格式为param = value。它可以用来设置以下参数:

- MASTER\_CONNECT\_RETRY = n 试图连接MySQL主服务器的各次尝试之间的等待时间间隔,以秒为计算单位。
- MASTER\_HOST = 'host\_name' MySQL主服务器所在的主机名。
- MASTER\_LOG\_FILE = 'file\_name' 主服务器的二进制变更日志的文件名,从服务器将使

用这个文件去建立镜像。

- MASTER\_LOG\_POS = *n* 主服务器日志文件中的某个位置，从服务器将从这个位置开始或者（在中断后）继续建立镜像。
- MASTER\_PASSWORD = '*pass\_val*' 连接主服务器时使用的口令。
- MASTER\_PORT = *n* 连接主服务器时使用的端口号。
- MASTER\_USER = '*user\_name*' 连接主服务器时使用的用户名。
- RELAY\_LOG\_FILE = '*file\_name*' 从服务器的中继日志的文件名。
- RELAY\_LOG\_POS = *n* 从服务器中继日志文件里的当前读写位置。

除主机名或端口号以外，只有明确给出的参数才会发生改变。改变主机或者端口号通常意味着你将换用一个不同的MySQL主服务器，所以主服务器的二进制变更日志和该日志的读写位置将被隐含地分别设置为空字符串和零。

CHANGE MASTER最早出现于MySQL 3.23.23版本。RELAY\_LOG\_FILE和RELAY\_LOG\_POS参数最早出现于MySQL 4.0.2版本（在此之前，从服务器中继日志文件的概念不存在）。

#### D.1.7 CHECK TABLE

语法：CHECK { TABLE | TABLES } *tbl\_name* [, *tbl\_name* ] . . . [ *options* ]

检查数据表有无错误。这条命令适用于MyISAM数据表和MySQL 3.23.39及以后版本中的InnoDB数据表。它要求你必须拥有各有关数据表上的SELECT权限。

*options*（如果给出的话）是由下列选项中的一个或者多个（彼此不以逗号分隔）组成的一个参数表：

- CHANGED 只对上次检查后又发生过修改或者没有被正常关闭的数据表进行检查。
- EXTENDED 进行扩展检查，以确保数据表完全没有错误。比如说，它将检查每个索引中的每个键字是否都指向一个数据行。这个选项的执行速度相当慢。
- FAST 只对没有被正常关闭的数据表进行检查。
- MEDIUM 对索引进行检查、扫描数据行以检查它们是否有错误、进行校验和验证。如果没有在*options*部分给出任何选项，就以此为默认值。
- QUICK 不扫描数据行，只检查索引。

CHECK TABLE的操作结果将返回如下所示的输出报告：

```
mysql> CHECK TABLE t;
+-----+-----+-----+-----+
| Table | Op    | Msg_type | Msg_text |
+-----+-----+-----+-----+
| test.t | check | status   | OK       |
+-----+-----+-----+-----+
```

ANALYZE TABLE、OPTIMIZE TABLE、REPAIR TABLE等语句所返回的信息也是这种格式。在上面的输出报告里，输出列Table给出了数据表的名字；输出列Op是所进行的操作，其可

取值为check、analyze、optimize或repair。输出列Msg\_type和Msg\_text则是相应的操作结果。

CHECK TABLE语句最早出现于MySQL 3.23.13版本，但在MySQL 3.23.25之前无法工作在Windows系统上。QUICK、FAST和MEDIUM选项分别始见于MySQL 3.23.16、3.23.23和3.23.31版本。在从MySQL 3.23.15到3.23.25版本里，这个语句每次只允许给出一个选项，而且必须以“TYPE =”的形式给出；在3.23.25之后的版本里，“TYPE =”被弃用，而且允许同时给出多个选项。

### D.1.8 COMMIT

**语法：**COMMIT

把当前事务中的各条语句所做出的修改正式写入数据库，使这些修改体现在此后的数据库里。COMMIT语句只能用在支持事务处理的数据表类型上。（如果是不支持事务处理的数据表类型，每一条语句所做出的修改会在该语句执行完毕后被正式写入数据库并生效。）

如果事先没有通过BEGIN语句或者通过把AUTOCOMMIT变量设置为0的办法把自动提交模式关闭掉，COMMIT语句将不起作用。

下面这些语句将隐含地结束并提交当前事务，就好像明确地发出了一条COMMIT语句一样：

```
ALTER TABLE
BEGIN
CREATE INDEX
DROP DATABASE
DROP INDEX
DROP TABLE
LOAD MASTER DATA
LOCK TABLES
RENAME TABLE
SET AUTOCOMMIT = 1
TRUNCATE TABLE
UNLOCK TABLES (如果数据表当前处于锁定状态的话)
```

COMMIT语句最早出现于MySQL 3.23.14版本。

### D.1.9 CREATE DATABASE

**语法：**CREATE DATABASE [ IF NOT EXISTS ] db\_name  
[ [ DEFAULT ] CHARACTER SET charset ]

以给定名称创建一个数据库。如果你没有创建该数据库的权限，这条语句将执行失败并报告出错。一般来说，如果你打算创建的数据库已经存在，这条语句通常会执行失败并报告出错；但如果你给出了IF NOT EXISTS子句，数据库将不会被创建，这条语句也不会报告出错。这个子句最早出现于MySQL 3.23.12版本。

DEFAULT CHARACTER SET子句最早出现于MySQL 4.0版本，可以用这个子句来设定数据库的默认字符集属性。charset可以是某个字符集的名字；也可以是关键字DEFAULT，表示该

数据库里的数据表都将使用服务器的当前字符集作为其默认字符集。数据库的各种属性都存放在对应于数据库目录中的db.opt文件里。

#### D.1.10 CREATE FUNCTION

语法: CREATE [ AGGREGATE ] FUNCTION function\_name  
 RETURNS { STRING | REAL | INTEGER }  
 SONAME 'shared\_library\_name'

把一个UDF (user-defined function, 用户定义函数) 加载到mysql数据库的func数据表里。*function\_name*是UDF的名字 (即你在SQL语句里使用这个函数时给出的函数名)。RETURNS后面的关键字指定了函数的返回值类型。字符串'*shared\_library\_name*'给出了一个路径名, 该UDF函数的可执行代码就包含在这个路径名所指定的文件里。

AGGREGATE关键字 (如果给出的话) 表明该UDF函数是一个类似于SUM()或MAX()函数的统计类函数; 这个关键字最早出现于MySQL 3.23.5版本。

CREATE FUNCTION语句要求MySQL服务器必须被编译为动态链接形式的二进制可执行代码 (不能是静态的二进制可执行代码), 这是因为UDF机制要求进行动态链接。UDF函数的编写方法详见*MySQL Reference Manual* (MySQL参考手册)。

#### D.1.11 CREATE INDEX

语法: CREATE [ UNIQUE | FULLTEXT ] INDEX index\_name  
 ON tbl\_name ( index\_columns )

在数据表*tbl\_name*里增加一个名为*index\_name*的索引。这条语句分别相当于ALTER TABLE ADD INDEX (如果在这条语句里既没有给出UNIQUE, 也没有给出FULLTEXT的话)、ALTER TABLE ADD UNIQUE (如果在这条语句里给出了关键字UNIQUE的话) 或ALTER TABLE ADD FULLTEXT (如果在这条语句里给出了关键字FULLTEXT的话), 详细讨论请参见前面对ALTER TABLE语句的描述。CREATE INDEX语句不能用来创建PRIMARY KEY——必须使用ALTER TABLE语句来创建它。

在需要给同一个数据表创建多个索引的时候, 最好直接使用一条ALTER TABLE语句来完成这项工作——只需使用一条语句就能把它们全都创建出来, 这要比使用多条CREATE INDEX语句去逐个创建索引的做法要简单快捷。

CREATE INDEX语句只能用在MySQL 3.22及以后的版本里。用来创建FULLTEXT索引的选项最早出现于MySQL 3.23.23版本。

#### D.1.12 CREATE TABLE

语法: CREATE [ TEMPORARY ] TABLE [ IF NOT EXISTS ] tbl\_name  
 (create\_definition,...)  
 [table\_options]  
 [[IGNORE | REPLACE] [AS] select\_statement]



```

create_definition:
{
  col_declaration [reference_definition]
  | [CONSTRAINT symbol] PRIMARY KEY (index_columns)
  | [CONSTRAINT symbol] UNIQUE [INDEX | KEY] [index_name] (index_columns)
  | {INDEX | KEY} [index_name] (index_columns)
  | FULLTEXT [INDEX | KEY] [index_name] (index_columns)
  | [CONSTRAINT symbol] FOREIGN KEY [index_name] (index_columns)
    [reference_definition]
  | [CONSTRAINT symbol] CHECK (expr)
}

col_declaration:
col_name col_type
[NOT NULL | NULL] [DEFAULT default_value]
[AUTO_INCREMENT] [PRIMARY KEY] [UNIQUE [KEY]]
[COMMENT 'string']

reference_definition:
REFERENCES tbl_name (index_columns)
[ON DELETE reference_action]
[ON UPDATE reference_action]
[MATCH FULL | MATCH PARTIAL]

reference_action:
{RESTRICT | CASCADE | SET NULL | NO ACTION | SET DEFAULT}

```

CREATE TABLE语句将在当前数据库里创建一个名为`tbl_name`的新数据表。但如果数据表的名字是以`db_name.tbl_name`的形式给出的，数据表就将创建在指定的数据库里。

如果给出了TEMPORARY关键字，这条语句就将创建一个临时数据表。临时数据表将在当前客户连接结束（不管是正常结束还是非正常结束）或者你用DROP TABLE语句丢弃它的时候“消失”；只有创建临时数据表的客户（程序）才能看到它，其他客户（程序）是不会察觉到还存在着一个这样的数据表的。

一般说来，如果你打算创建的数据表已经存在，这条语句将执行失败并报告出错。但这里又有两个例外。首先，如果你给出了IF NOT EXISTS子句，数据表将不会被创建，这条语句也不会报告出错。其次，如果你给出了TEMPORARY关键字而已经存在的同名数据表不是一个临时数据表，这条语句就会创建出一个临时数据表来。在这个临时数据表存在期间，原来那个名为`tbl_name`的数据表将自动“隐藏”起来而不让创建该临时数据表的那个客户（程序）看到，但其他客户（程序）包括你本人的下一次客户会话仍能看到原来的那个数据表；如果你在这次客户会话里用DROP TABLE语句丢弃了这个临时数据表或者把它重新命名为另外一个名字，就可以再次看到原已存在的那个数据表了。

`create_definition`部分是你准备在这个数据表里创建的各数据列和索引的定义，但如果新数据表是通过子查询机制（即你在CREATE TABLE语句里给出了`select_statement`部分，它可以是

任意形式的SELECT查询语句)而创建出来的,这个`create_definition`部分就可以省略。`table_options`子句允许你为新数据表设定各种属性。如果新数据表是通过子查询机制而创建出来的,新数据表将使用那个SELECT语句(即子查询)所返回的结果集来创建。关于这几个子句的详细讨论见随后的几个小节。

IF NOT EXISTS子句、`table_options`子句以及通过子查询机制来创建新数据表的机制最早出现于MySQL 3.23版本。TEMPORARY数据表最早出现于MySQL 3.23.2版本。

#### 1. 数据列和索引的定义

CREATE TABLE语句的`create_definition`部分可以是一个数据列或索引定义、一个FOREIGN KEY子句或者一个CHECK子句。引入CHECK子句的目的是为了使MySQL能够与其他数据库系统保持兼容,但它对MySQL数据库里的数据表不起作用,MySQL在遇到CHECK子句的时候会忽略之;FOREIGN KEY子句也大致如此,但InnoDB数据表上的FOREIGN KEY子句是起作用的。

`create_definition`部分中的数据列定义将以一个数据列名称`col_name`和该数据列的类型`col_type`开始,后面通常还会有几个可选的关键字。数据列的类型可以是附录B里列出的任何一种数据列类型。每种数据列类型都有一些特有的属性,对这些属性的详细讨论请参见附录B。允许出现在数据列类型`col_type`之后的其他可选关键字如下所示:

- NULL或NOT NULL

用来表明数据列是否允许包含NULL值。如果这两个关键字都没有给出,则将以NULL为默认设置。

- DEFAULT *default\_value*

用来设定数据列的默认值。注意,不能为BLOB和TEXT类型的数据列设置默认值。数据列的默认值必须是一个常数,它可以是一个数值、一个字符串或者NULL值。

如果没有给数据列设置默认值,就将由MySQL替它安排一个。对于那些允许包含NULL值的数据列,MySQL将把NULL值安排为它的默认值;对于那些不允许包含NULL值的数据列,MySQL将根据以下规则来安排它的默认值:

- 对于数值类型的数据列(不包括AUTO\_INCREMENT数据列),其默认值将是0。对于AUTO\_INCREMENT数据列,其默认值将是下一个序列编号值。
- 对于日期和时间类型的数据列(不包括TIMESTAMP数据列),其默认值将是有关类型的“零值”(例如,DATE数据列的零值是'0000-00-00')。对于TIMESTAMP数据列,其默认值将是当前日期加当前时间(数据表里的第一个TIMESTAMP数据列)或零值(数据表里的其他TIMESTAMP数据列)。
- 对于字符串类型的数据列(不包括ENUM数据列),其默认值将是空字符串。对于ENUM数据列,其默认值将是枚举集合中的第一个元素。

- AUTO\_INCREMENT

这个关键字只能用在整数类型的数据列上。AUTO\_INCREMENT数据列的特殊之处在于:当你向其中插入一个NULL值时,实际被插入的数据值将是有关序列的下一个编号值(它通常等于该数据列里的当前最大编号值再加上一个1)。

在默认的情况下,AUTO\_INCREMENT数据列的实际取值将从1开始编号。对于MyISAM

数据表（以及MySQL 4.1版本以后的HEAP数据表），可以通过数据表选项 `AUTO_INCREMENT = n` 来明确地设定一个起始编号值。`AUTO_INCREMENT`数据列必须被同时声明为 `UNIQUE` 索引或 `PRIMARY KEY`，还必须同时被声明为 `NOT NULL`。每个数据表最多只能有一个 `AUTO_INCREMENT` 数据列。

- **PRIMARY KEY**

用来表明数据列是一个 `PRIMARY KEY`。`PRIMARY KEY` 数据列必须同时被声明为 `NOT NULL`。

- **UNIQUE [ KEY ]**

用来表明数据列是一个 `UNIQUE` 索引。这个属性最早出现于MySQL 3.23版本。

- **COMMENT 'string'**

用来表明字符串 `'string'` 是对数据列的描述性注释。在MySQL 4.1之前的版本里，这个属性将被MySQL的语法分析器忽略。在MySQL 4.1及以后的版本里，MySQL服务器则会记住这个字符串并能把它显示在 `SHOW CREATE TABLE` 或 `SHOW FULL COLUMNS` 命令的输出报告里。

`PRIMARY KEY`、`UNIQUE`、`INDEX`、`KEY`、`FULLTEXT` 等子句的用途是创建各种索引。`PRIMARY KEY` 和 `UNIQUE` 子句所创建的索引不允许包含彼此相同的值（也就是所谓的惟一化索引）。`INDEX` 和 `KEY` 互为同义词，它们所创建的索引允许包含彼此相同的值（也就是所谓的非惟一化索引）。这几个子句所建立的索引将建立在相应的 `index_columns` 部分所列举出来的数据列（它们都必须真正存在于数据表 `tbl_name` 里）上；如果涉及多个数据列，则必须以逗号把它们分隔开。对于 `CHAR` 或 `VARCHAR` 数据列，可以只索引数据列的一个前缀，即通过 `col_name(n)` 语法来表明你只需要对数据列里的各项数据值的前 `n` 个字节进行索引。（注意，这里有一个例外——InnoDB数据表不允许对数据列前缀进行索引）。对于 `BLOB` 或 `TEXT` 数据列，必须给出一个前缀值，不能对整个数据列进行索引。但 `FULLTEXT` 索引里的数据列前缀值（如果给出的话）却将被忽略。如果没有给出索引名 `index_name`，MySQL将根据第一个被索引的数据列名字自动选定一个索引名。

`FULLTEXT` 索引只能创建在MyISAM数据表里，并且只能创建在 `TEXT` 数据列以及非 `BINARY CHAR` 和 `VARCHAR` 数据列上。

ISAM数据表或MySQL 4.0.2之前的HEAP数据表里的被索引数据列必须声明为 `NOT NULL`。`PRIMARY KEY` 数据列永远必须被声明为 `NOT NULL`。

## 2. 数据表选项

`table_options` 子句最早出现于MySQL 3.23版本（有些选项始见于更晚的版本里，详见下面的具体说明）。数据表选项可以包括以下清单里的一个或者多个选项；如果需要同时给出多个选项，就必须以逗号把它们分隔开。如无特殊说明，下面这些选项将适用于所有的数据表类型。

- **AUTO\_INCREMENT = n**

为 `AUTO_INCREMENT` 数据列设定一个起始编号值。这个选项只能用在MyISAM数据表以及从MySQL 4.1版本开始的HEAP数据表里。

- **AVG\_ROW\_LENGTH = *n***

数据表中的数据行平均长度。对于MyISAM数据表，MySQL将根据AVG\_ROW\_LENGTH和MAX\_ROWS的乘积来确定数据文件的最大长度。MyISAM数据表处理程序内部使用的数据行指针的宽度可以是1~8个字节，这个指针的默认宽度足以允许你创建出4GB的数据表来。如果你需要用到更大的数据表（并且你的操作系统也支持如此之大的文件的话），就可以利用AVG\_ROW\_LENGTH和MAX\_ROWS选项来加大MyISAM数据表处理程序内部使用的数据行指针的宽度。这两个选项值的乘积越大，MyISAM数据表处理程序内部使用的数据行指针的宽度也就越大。如果数据表本身的尺寸比较小，这种做法就不见得能替你节省下多少空间来，但如果你有很多小尺寸的数据表，累积起来的节省就很可观了。

- **[ DEFAULT ] CHARACTER SET *charset***

为数据表指定一个默认字符集。*charset*可以是某个字符集的名字；也可以是关键字DEFAULT，即数据表将使用数据库的当前字符集（如果你为这个数据库指定了当前字符集的话）或者服务器的当前字符集作为其默认字符集。如果在定义这个数据表的某个字符串数据列时没有明确地表明它将使用哪一个字符集，该数据列里的数据值就将使用本选项所确定的字符集。在下面的例子里，数据列c1将使用sjis字符集，而c2则将使用ujis字符集：

```
CREATE TABLE t
(
    c1 CHAR(50) CHARACTER SET sjis,
    c2 CHAR(50)
) CHARACTER SET ujis;
```

这个选项还将影响到你以后使用ALTER TABLE语句对字符串数据列的修改操作——如果没有在ALTER TABLE操作中明确地指定一个字符集，有关的字符串数据列就将使用这个选项所指定的字符集。

CHARACTER SET选项最早出现于MySQL 4.1版本。它允许以任何同义格式给出，比如说，下面这几项设置就是等价的：

```
CHARACTER SET charset
CHARSET = charset
CHARSET charset
```

这些同义格式可以用在需要对字符集进行设定的任何场所，比如数据列定义、CREATE DATABASE和ALTER DATABASE语句里。

- **CHECKSUM = { 0 | 1 }**

如果这个选项被设置为1，MySQL就将为数据表里的每一个数据行生成一个校验和并进行相应的检查。校验和的生成和检查工作会给数据表的更新操作稍微增加一点儿开销，但能提高数据表检查操作（即CHECK TABLE语句）的效率。这个选项只适用于MyISAM数据表。

- **COMMENT = '*string*'**

给数据表增加注释。注释的长度最大为60个字符。可以通过SHOW CREATE TABLE和SHOW TABLE STATUS语句来查看注释。



- **DATA DIRECTORY = 'dir\_name'**

这个选项只适用于MyISAM数据表。它用来规定数据文件（即.MYD文件）必须写到指定的目录里去。'dir\_name'必须是一个完整的路径名。这个选项最早出现于MySQL 4.0版本，但只能工作在服务器未使用--skip-symlink选项而启动的场合里。在某些操作系统（比如Mac OS X、FreeBSD或BSDI）上，symlink无法配合线程机制工作，因而通常会被默认地禁用掉。

- **DELAY\_KEY\_WRITE = { 0 | 1 }**

如果这个选项被设置为1，索引缓存区里的内容将被定期写到磁盘上而不是在每个插入操作完成之后立刻进行这种写操作。这个选项只适用于MyISAM数据表。

- **INDEX DIRECTORY = 'dir\_name'**

这个选项只适用于MyISAM数据表。它用来规定索引文件（即.MYI文件）必须写到指定的目录里去。'dir\_name'必须是一个完整的路径名。这个选项最早出现于MySQL 4.0版本，但只能工作在服务器未使用--skip-symlink选项而启动的场合里。在某些操作系统（比如Mac OS X、FreeBSD或BSDI）上，symlink无法配合线程机制工作，因而通常会被默认地禁用掉。

- **INSERT\_METHOD = { NO | FIRST | LAST }**

这个选项用来设定MERGE数据表将如何插入数据行。NO表示根本不允许插入数据行，FIRST或LAST则表示数据行将被插入到组成这一MERGE数据表的第一个或最后一个MyISAM数据表里去。这个选项最早出现于MySQL 4.0版本。

- **MAX\_ROWS = n**

打算存放到这个数据表里去的最大数据行数。这个选项的具体用途和用法见上面对AVG\_ROW\_LENGTH选项的介绍。这个选项只适用于MyISAM数据表。

- **MIN\_ROWS = n**

打算存放到这个数据表里去的最小数据行数。这个选项主要用于HEAP数据表，它能向HEAP处理程序提供一些内存优化方面的提示。

- **PACK\_KEYS = { 0 | 1 | DEFAULT }**

这个选项控制着MyISAM和ISAM数据表中的索引压缩功能，即是否需要相似的索引值进行压缩和如何压缩。索引压缩功能会增加数据表更新（写）操作的开销，但能改善检索（读）操作的性能。0表示不进行压缩；1表示对字符串（即CHAR或VARCHAR）值以及（MyISAM数据表中的）数值型索引值进行压缩；DEFAULT（始见于MySQL 4.0版本）表示只对长字符串数据列进行压缩。

- **PASSWORD = 'string'**

设定一个用来加密数据表说明文件（.frm文件）的口令。如果你没有与MySQL数据库系统的运营方（你是使用方）签订有关的服务支持合同，这个选项通常没有什么作用。

- **RAID\_TYPE = { 1 | STRIPED | RAID0 }**

**RAID\_CHUNKS = n**



`RAID_CHUNKSIZE = n`

这些选项最早出现于MySQL 3.23.12版本。它们的用途是扩展MyISAM数据表的有效长度。但如果你在编译MySQL软件时没有使用`--with-raid`选项，这些选项将不起作用。

`RAID_TYPE`选项的默认设置值是`STRIPED`；另两个设置值其实是`STRIPED`的同义词。`RAID_CHUNKS`和`RAID_CHUNKSIZE`选项控制着MySQL为MyISAM数据表里的数据分配存储空间的方式。MySQL将在数据库目录下创建多个（数量由`RAID_CHUNKS`选项值决定）下级目录并在每个下级目录里分别创建一个名为`tbl_name.MYD`的数据文件。当往数据表里添加数据行时，MySQL将先把数据行写到第一个下级目录的文件里去，等它写满之后，再依次前进到下一个下级目录。各下级目录里的文件尺寸由`RAID_CHUNKSIZE`选项值控制，它的计量单位是MB（1024K字节）。那些下级目录的名字按十六进制数字00、01……排列。比如说，如果`RAID_CHUNKS`选项的值是256，`RAID_CHUNKSIZE`选项的值是1000，那么MySQL服务器将建立256个下级目录——它们的名字从00一直到ff，并将在每一个下级目录的文件里写满1000MB字节。

- `ROW_FORMAT = { DEFAULT | FIXED | DYNAMIC | COMPRESSED }`

这个选项只适用于MyISAM数据表，它用来设定数据行的存储方式。这个选项最早出现于MySQL 3.23.6版本。

- `TYPE = { ISAM | MYISAM | MERGE | HEAP | BDB | INNODB }`

用来设定数据表的存储格式。对各种数据表存储格式的讨论详见第3.3.1节。从MySQL 3.23版本开始，如果你没有在对服务器进行编译时另做配置、也没有在启动服务器时使用`--default-table-type`选项另行指定，数据表的默认格式就将是MyISAM。`MRG_MYISAM`、`BERKELEYDB`和`INNODB`分别是`MERGE`、`BDB`和`INNODB`的同义词。如果你在创建数据表时指定的数据表类型是合法的，但MySQL服务器里却没有配置该类型的处理程序，MySQL将使用默认的类型来创建数据表。如果在这个选项处给出了一个非法值，操作将失败并报告出错。这个选项最早出现于MySQL 3.23版本；在此之前，`CREATE TABLE`语句所创建出来的数据表只有ISAM一种格式。

- `UNION = ( tbl_list )`

这个选项只适用于`MERGE`数据表，它列出了构成`MERGE`数据表的各个MyISAM数据表。

### 3. 子查询机制

如果`CREATE TABLE`语句包含`select_statement`子句（这个子句本身可以是一条任意形式的`SELECT`查询命令）部分，MySQL就将使用该子句所返回的结果集来创建新数据表。对于那些会导致惟一化索引出现重复键值的数据行，MySQL将按以下原则处理：如果给出了`IGNORE`关键字，则忽略（不插入）后出现的数据行；如果给出了`REPLACE`关键字，则后出现的数据行将替换掉先出现的数据行；如果这两个关键字都没有给出，`CREATE TABLE`语句将执行失败并报告出错。

### 4. 外键支持

InnoDB数据表处理程序提供了外键支持机制。外键在有关数据表之间构成了父、子层次关系。外键必须通过在子数据表里使用关键字`FOREIGN KEY`进行定义，在这个关键字的后面是一

个可选的索引名称，然后是构成这一外键的数据列清单，最后是一个REFERENCES定义。外键的索引名称即使给出也会被忽略。在外键的REFERENCES定义里，需要列出外键所用到的父数据表和其中的数据列，并规定父表中的记录被删除时应该采取什么样的动作。InnoDB目前实现的动作有CASCADE（删除子表中的相关记录）和SET NULL（把子表中的相关记录设置为NULL）；RESTRICT、NO ACTION和SET DEFAULT等动作目前还未实现，MySQL语法分析器在遇到它们的时候会忽略之。

REFERENCES定义中的ON UPDATE和MATCH子句也尚未得到实现，MySQL语法分析器在遇到它们的时候会忽略之。（如果你是在非InnoDB类型的数据表里定义外键的，那么整个定义都将被忽略。）

下面是一些演示CREATE TABLE语句用法的示例。

首先，我们来创建一个由id、last\_name和first\_name三个数据列构成的customer数据表。其中，id数据列将被定义为PRIMARY KEY，而last\_name和first\_name数据列则将共同构成一个索引：

```
CREATE TABLE customer
(
    id          SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,
    last_name   CHAR(30) NOT NULL,
    first_name  CHAR(20) NOT NULL,
    PRIMARY KEY (id),
    INDEX (last_name, first_name)
);
```

接下来，再创建一个临时数据表。为加快其处理速度，把这个临时数据表定义为一个HEAP数据表（HEAP数据表只存在于内存里而不会被写到磁盘上）：

```
CREATE TEMPORARY TABLE tmp_table
(id MEDIUMINT NOT NULL UNIQUE, name CHAR(40))
TYPE = HEAP;
```

下面这条语句创建的新数据表其实就是老数据表的一份拷贝：

```
CREATE TABLE prez_copy SELECT * FROM president;
```

下面这条语句创建的新数据表其实就是老数据表部分内容的一份拷贝：

```
CREATE TABLE prez_alive SELECT last_name, first_name, birth
FROM president WHERE death IS NULL;
```

如果在通过子查询机制创建新数据表时还给出了一些创建定义，这些定义将在新数据表被创建出来且已填充完有关数据之后施加到新数据表上。比如说，可以像下面这样把新数据表中的某个数据列定义为PRIMARY KEY：

```
CREATE TABLE new_tbl (PRIMARY KEY (a)) SELECT a, b, c FROM old_tbl;
```

从MySQL 4.1版本开始，还可以对新数据表中的数据列再次进行定义，新定义将覆盖各有关数据列从老数据表的查询结果集里沿袭而来的默认属性，如下所示：

```
CREATE TABLE new_tbl (a INT NOT NULL AUTO_INCREMENT, b DATE, PRIMARY KEY (a))
SELECT a, b, c FROM old_tbl;
```

## D.1.13 DELETE

```

语法: DELETE [ LOW_PRIORITY ] [ QUICK ] FROM tbl_name
      [ WHERE where_expr ] [ ORDER BY . . . ] [ LIMIT n ]
DELETE [ LOW_PRIORITY ] [ QUICK ] tbl_name [, tbl_name ] . . .
FROM tbl_name [, tbl_name ] . . .
[ WHERE where_expr ]
DELETE [ LOW_PRIORITY ] [ QUICK ] FROM tbl_name [, tbl_name ] . . .
USING tbl_name [, tbl_name ] . . .
[ WHERE where_expr ]

```

DELETE语句的第一种形式用来从数据表`tbl_name`里把那些符合WHERE子句所给条件的数据行全部删除掉,如下所示:

```

DELETE FROM score WHERE event_id = 14;
DELETE FROM member WHERE expiration < CURDATE();

```

如果WHERE子句被省略,则数据表中的全体数据行都将被删除!

LOW\_PRIORITY选项(如果给出的话)将使DELETE语句被延缓到没有客户(程序)在对数据表进行写操作时才得以执行。LOW\_PRIORITY选项最早出现于MySQL 3.22.5版本。

对于MyISAM数据表,给出QUICK选项将加快语句的执行速度;MyISAM处理程序将不进行通常要执行的索引树叶结点合并工作。QUICK选项最早出现于MySQL 3.23.25版本。

LIMIT子句(如果给出的话)将把DELETE语句将要删除的数据行的最大个数限制为 $n$ 个。LIMIT子句最早出现于MySQL 3.22.7版本。

ORDER BY子句(如果给出的话)将使DELETE语句先对结果集进行排序,然后再进行删除操作。把它与LIMIT子句结合使用,就能对将要删除哪些数据行的问题进行更精确的控制。ORDER BY子句最早出现于MySQL 4.0.0版本,它的具体格式和语法与SELECT语句中的情况完全相同。

在一般情况下,DELETE语句会把它实际删除了的数据行个数作为返回值。但不带WHERE子句的DELETE语句却存在着这样一个问题:在MySQL 4之前的版本里,服务器会把这个特殊情况的具体做法优化为先丢弃整个数据表、再重新建立一个结构完全相同的新数据表,而不是一个一个地依次删除全体数据行。这种做法速度很快,但返回的数据行计数值却可能是0。如果你想知道到底删除了多少个数据行,就需要给出一条能够匹配全体数据行的WHERE子句,如下所示:

```
DELETE FROM tbl_name WHERE 1;
```

但这种一个一个地依次删除全体数据行的做法将会大大降低数据行删除操作的速度。

如果你不需要知道到底删除了多少个数据行,还可以利用TRUNCATE TABLE语句来迅速清空整个数据表。

DELETE语句的第二、第三种形式特别适用于想从多个数据表一次性地删除有关数据行的情

况。它们使你能够根据各有关数据表之间的关联关系来确定需要删除哪些数据行。DELETE语句的这两种形式分别始见于MySQL 4.0.0和4.0.2版本。在这两种形式的DELETE语句里，数据表的名字既可以写成`tbl_name`的形式，也可以写成`tbl_name.*`（这种写法是为了与ODBC保持兼容）的形式。

举个例子：假设需要把数据表`t1`里的`id`值与数据表`t2`里的`id`值相等的数据行全都删除掉。下面是采用第一种多数据表语法来完成这一工作的DELETE语句：

```
DELETE t1 FROM t1, t2 WHERE t1.id = t2.id;
```

下面是采用第二种多数据表语法来完成这一工作的DELETE语句：

```
DELETE FROM t1 USING t1, t2 WHERE t1.id = t2.id;
```

#### D.1.14 DESCRIBE

语法： { DESCRIBE | DESC } `tbl_name` [ `col_name` | 'pattern' ]  
 { DESCRIBE | DESC } `select_statement`

DESCRIBE语句的第一种形式（带数据表名称参数`tbl_name`）将产生与SHOW COLUMNS语句完全相同的输出报告，详细情况请参见后面内容里的SHOW条目。在这种语法里，如果还给出了某个数据列的名字（`col_name`），DESCRIBE语句的输出报告中就将只包含关于该数据列的信息；如果还给出了一个字符串（'pattern'），DESCRIBE语句就会把它视为一个LIKE操作符匹配模式，它的输出报告里就将包含所有与这个模式相匹配的数据列的有关信息。

比如说，下面这条语句将把关于`president`数据表里的`last_name`数据列的描述信息显示出来：

```
DESCRIBE president last_name;
```

如果你想查看关于`president`数据表里的`last_name`和`first_name`两个数据列的描述信息，可以使用下面这条语句：

```
DESCRIBE president '%name';
```

DESCRIBE语句的第二种形式（带一个SELECT查询语句）其实是EXPLAIN语句的同义词，详细情况请参见后面内容里的EXPLAIN条目。（MySQL里的DESCRIBE和EXPLAIN语句其实是功能完全相同的同义词，但人们通常习惯于使用DESCRIBE语句来获得关于数据表的描述信息、使用EXPLAIN语句来获得关于SELECT查询语句表的描述信息。）

#### D.1.15 DO

语法：DO `expr` [, `expr` ] . . .

对表达式进行求值，但不返回求值结果。因为不存在需要对结果集进行处理的问题，所以DO语句往往要比SELECT语句更适合用来对表达式进行求值。DO语句的常见用途有两种，一是用来对变量进行赋值；二是用来调用某些你只关心其“副作用”而不太关心其返回值的函数，如下所示：

```
DO @sidea := 3, @sideb := 4, @sidec := SQRT(@sidea*@sidea+@sideb*@sideb);  
DO RELEASE_LOCK('mylock');
```

DO语句最早出现于MySQL 3.23.47版本。

#### D.1.16 DROP DATABASE

**语法:** DROP DATABASE [ IF EXISTS ] db\_name

丢弃（即删除）指定的数据库。如果丢弃了数据库，其中的内容就永远找不回来了，所以在打算使用这条语句的时候一定要三思而后行。如果打算删除的数据库不存在（除非你给出了IF EXISTS关键字）或者你不具备必要的权限，这条语句将执行失败并报告出错。IF EXISTS关键字的作用是抑制（即不显示）MySQL在你试图删除一个并不存在的数据库时所给出的出错信息。IF EXISTS关键字最早出现于MySQL 3.22.2版本。

一个数据库就是MySQL数据目录里的一个目录，而这个目录里的非数据表文件是不会被DROP DATABASE语句删除的。在这种情况下，数据库目录本身将不会被删除，所以你仍会在SHOW DATABASE语句的输出报告里看到这个数据库的名字。

#### D.1.17 DROP FUNCTION

**语法:** DROP FUNCTION function\_name

丢弃（即删除）此前用CREATE FUNCTION语句加载的UDF（user-defined function，用户定义函数）。

#### D.1.18 DROP INDEX

**语法:** DROP INDEX index\_name ON tbl\_name

从数据表tbl\_name里丢弃（即删除）名为index\_name的索引。MySQL将把这条语句当做一条ALTER TABLE DROP INDEX语句来处理，详细情况请参见前面内容里的ALTER TABLE条目。注意，DROP INDEX语句不能用来丢弃PRIMARY KEY，必须使用相应的ALTER TABLE语句来完成这一工作。

DROP INDEX语句只能用在MySQL 3.22及以后的版本里。

#### D.1.19 DROP TABLE

**语法:** DROP TABLE [ IF EXISTS ] tbl\_name [, tbl\_name ] . . . [ RESTRICT | CASCADE ]

从它们所在的数据库里丢弃（即删除）指定的数据表。如果给出了IF EXISTS关键字，MySQL就不会在你试图丢弃一个并不存在的数据表时报告出错。IF EXISTS关键字最早出现于MySQL 3.22.2版本。

RESTRICT和CASCADE关键字不起作用，它们的用途是使MySQL与其他数据库系统的代码保持兼容，但MySQL会忽略它们。这两个关键字最早出现于MySQL 3.23.29版本。



## D.1.20 EXPLAIN

语法: `EXPLAIN tbl_name [ col_name | 'pattern' ]`  
`EXPLAIN select_statement`

这条语句的第一种形式相当于一條DESCRIBE *tbl\_name*语句, 详细情况请参见前面内容里的DESCRIBE条目。

EXPLAIN语句的第二种形式能够让我们了解到MySQL将如何去执行出现在关键字EXPLAIN之后的SELECT语句, 如下所示:

```
EXPLAIN SELECT score.* FROM score, event
WHERE score.event_id = event.event_id AND event.event_id = 14;
```

EXPLAIN语句的输出由包含以下数据列的一个或者多个输出行构成:

- **table** 各输出行里的信息是关于哪个数据表的。
- **type** MySQL将进行的关联操作的类型。这些类型(按限制性由大到小的顺序排列)包括: system、const、eq\_ref、ref、range、index和ALL。排在前面的类型有着更严格的限制, 即需要MySQL在检索过程中去进行检查的数据行相对要更少一些。
- **possible\_keys** MySQL认为在指定数据表(即名称出现在table输出行里的那个数据表)里对数据行进行检索时可能会用到的各有关索引的名字。如果这个输出列里的值是NULL, 则表明没有在数据表里找到这样的索引。
- **key** MySQL在指定数据表(即名称出现在table输出行里的那个数据表)里对数据行进行检索时实际用到的各有关索引的名字。如果这个输出列里的值是NULL, 则表明没有在数据表里找到这样的索引。
- **key\_len** 实际使用的索引键字的长度。如果MySQL实际使用的是有关索引的一个最左前缀(即只使用了单个索引项的前*n*个字节)的话, 这个输出列里的数字可能会小于单个索引项的总长度。
- **ref** MySQL用来与索引键值进行比较的值。如果这个输出列里的内容是单词const或'???' , 则表示比较对象是一个常数; 如果这个输出列里的内容是某个数据列的名称, 则表示有关的比较操作是一个数据列一个数据列地进行的。
- **rows** MySQL为完成这个查询而需要在各数据表里加以检查的数据行个数的估算值。这个输出列里的值的乘积就是这个查询所涉及的各有关数据表中的各有关数据行的各种可能组合的估算值。
- **Extra** 如果这个输出列里的内容是Using index或者是Only index(在较早的MySQL版本里), 则表明MySQL只需使用索引信息就能把有关信息从数据表里检索出来, 不需要对数据文件进行检查; 如果这个输出列里的内容是Using where, 则表明MySQL需要利用该SELECT语句中的WHERE子句才能把有关信息检索出来。

## D.1.21 FLUSH

语法: `FLUSH option [, option] . . .`

对MySQL服务器内部使用的各种缓存区进行刷新（指把缓存区里的内容写到磁盘或者从磁盘重新加载其原始内容）。下面是这条语句的`option`部分的各种可取值：

- **DES\_KEY\_FILE** 重新加载供**DES\_ENCRYPT()**和**DES\_DECRYPT()**函数用来完成加密/解密工作的DES密钥文件。这个选项最早出现于MySQL 4.0.1版本。
- **HOSTS** 刷新主机缓存区里的信息。
- **LOGS** 刷新日志文件；先关闭这些文件，再重新打开它们。
- **MASTER** 这个选项现已被重新命名为**RESET MASTER**，请使用新名字。
- **PRIVILEGES** 重新加载MySQL数据库系统的各种权限表。MySQL服务器在启动时会从磁盘上把各权限表文件的内容依次读入内存，如果通过**GRANT**或**REVOKE**命令修改了这些数据表，MySQL服务器将自动同步更新它们在磁盘和内存中的拷贝；但如果你是通过**INSERT**或**UPDATE**等语句来直接修改权限表的，就必须利用这个选项明确地让MySQL重新加载它们。这个选项还将把资源管理极限值重新设置为零，这一点类似于**USER\_RESOURCES**选项。
- **QUERY CACHE** 刷新查询缓存区以对之进行碎片整理，但不清除这个缓存区里的语句。（如果你想彻底清除这个缓存区，就需要使用**RESET QUERY CACHE**命令。）这个选项最早出现于MySQL 4.0.1版本。
- **SLAVE** 这个选项已被重新命名为**RESET SLAVE**，请参见**RESET**语句。
- **STATUS** 重新对各有关状态变量进行初始化。这个选项最早出现于MySQL 3.22.11版本。
- **TABLES [tbl\_name [,tbl\_name] ...]** 刷新指定的数据表缓存区。如果只给出了**TABLES**关键字而没有列出数据表的名字，服务器将先关闭、再重新打开数据表缓存区里的所有数据表。从MySQL 3.23.23版本开始，如果你只想刷新某一个或者某几个特定的数据表而不想刷新整个数据表缓存区，可以在**TABLES**关键字的后面把它们的名字（以逗号分隔）列举出来。也是从这个版本开始，**FLUSH TABLES**语句新增了一个同义词**FLUSH TABLE**。此外，如果查询缓存区也可操作，**FLUSH TABLES**语句还将刷新查询缓存区。
- **TABLES WITH READ LOCK** 先刷新所有数据库里的所有数据表，然后给它们加上一个读锁定，这个读锁定将一直作用到你发出一条**UNLOCK TABLES**语句为止。这条语句仍允许客户（程序）读取数据表里的内容，但将阻塞对这些数据表的修改（写）操作——如果你想把MySQL服务器在某个特定时刻的内容完整地备份下来，使用本选项将确保各有关数据表不会在你进行备份操作期间发生改变。当然，从客户的角度看，这意味着无法对数据表进行修改的时间被延长了。这个选项最早出现于MySQL 3.23.18版本。
- **USER\_RESOURCES** 把当前账户的各项资源管理极限值（比如**MAX\_QUERIES\_PER\_HOUR**等）重置为它们各自的初始值，这样，那些资源使用量已经达到有关极限值上限的账户将又可以使用了。这个选项最早出现于MySQL 4.0.2版本。

要想执行**FLUSH**语句，必须具备**RELOAD**权限。**FLUSH**语句最早出现于MySQL 3.22.9版本；它的部分选项始见于稍晚的版本，具体情况如上所示。

## D.1.22 GRANT

```

语法: GRANT priv_type [(column_list)] [, priv_type [(column_list)] ] ...
      ON {*. * | * | db_name.* | db_name.tbl_name | tbl_name}
      TO account [IDENTIFIED BY 'password']
          [, account [IDENTIFIED BY 'password'] ] ...
      [REQUIRE security_options]
      [WITH grant_or_resource_options]

```

这条语句将把其`priv_type`部分所列举的访问权限（如下表所示）授予一个或者多个MySQL用户。除必须单独使用的ALL权限外，如果需要列举多项权限，必须用逗号把它们分隔开。ALL是所有其他权限（但GRANT OPTION权限不包括在内）的组合，GRANT OPTION权限必须单独使用一条GRANT语句或者通过增加一个WITH GRANT OPTION子句来进行授予。

权限名称	该权限所允许的操作
ALTER	更改数据表和索引
CREATE	创建数据库和数据表
CREATE TEMPORARY TABLES	创建临时数据表
DELETE	删除数据表里的现有数据行
DROP	丢弃（删除）数据库和数据表
EXECUTE	执行存储过程（保留供今后使用）
FILE	读、写服务器主机上的文件
GRANT OPTION	把账户的权限授予其他账户
INDEX	创建或者丢弃索引
INSERT	往数据表里插入新数据行
LOCK TABLES	用LOCK TABLES语句明确地锁定数据表
PROCESS	查看关于在服务器里运行着的各有关线程的信息
REFERENCES	未使用（保留供今后使用）
RELOAD	重新加载各种权限表或者对各有关日志及缓存区进行刷新
REPLICATION CLIENT	查知主、从服务器的运行地点（即主、从服务器所在的主机名）
REPLICATION SLAVE	作为镜像从服务器
SELECT	对数据表里的现有数据行进行检索
SHOW DATABASES	发出SHOW DATABASES语句
SHUTDOWN	关闭MySQL服务器
SUPER	用KILL命令终止线程以及进行其他超级用户操作
UPDATE	对数据表里的现有数据行进行修改
ALL	所有操作（GRANT OPTION权限不包括在内）；ALL PRIVILEGES是本权限的同义词
USAGE	一个特殊的“无权限”权限

LOCK TABLES权限只能作用于你同时还拥有其SELECT权限的那些数据表上，但它允许放置任何种类（读、写等等）的数据锁而不仅限于读锁定。

你总是能查看或者终止属于你自己的线程，但SUPER权限允许你查看任何一个线程。

CREATE TEMPORARY TABLES、EXECUTE、LOCK TABLES、REPLICATION CLIENT、

REPLICATION SLAVE、SHOW DATABASES、SUPER等权限最早出现于MySQL 4.0.2版本。在MySQL 4.0.2之前的版本里，SHOW DATABASES、REPLICATION CLIENT和REPLICATION SLAVE、SUPER权限分别由SELECT、FILE、PROCESS权限控制。

ON子句负责设定有关权限的作用范围，如下表所示：

权限标识符	有关权限的作用范围
ON *.*	全局权限，其作用范围是所有数据库里的所有数据表
ON *	全局权限，若未指定默认数据库，其作用范围是所有数据库里的所有数据表；否则，其作用范围是当前数据库里的所有数据表
ON db_name.*	数据库级权限，其作用范围是指定数据库里的所有数据表
ON db_name.tbl_name	数据表级权限，其作用范围是指定数据表里的所有数据列
ON tbl_name	数据表级权限，其作用范围是默认数据库中指定数据表里的所有数据列

当某个数据表的名字出现在ON子句里的时候，还可以在GRANT语句的column\_list子句里列举一个或者多个数据列（它们彼此要用逗号分隔开），以把有关权限施加在指定的数据列上。但这种做法只适用于INSERT、REFERENCES、SELECT和UPDATE权限，而允许施加在数据列上的权限也仅限于这几种。

TO子句负责指定有关权限将被授予哪些MySQL账户。这些账户要求以'user\_name'@'host\_name'的格式给出，而且，在它们的后面，还可以通过可选的IDENTIFIED BY子句设定口令。如果不包含特殊字符，账户名中的user\_name和host\_name也可以不加单引号；但只要使用了单引号，就必须把它们分别放在两对单引号里。（比如说，账户名bill@%.com在加上引号之后必须写成'bill'@'%.com'，而不允许写成'bill@%.com'。）user\_name可以是一个用户名，也可以是一个空字符串（即''）；后一种情况的意思是把有关权限授予给一个匿名用户。host\_name允许是localhost、主机名、IP地址或者用来匹配域名或网络号的匹配模式。允许使用的匹配模式字符是“%”和“\_”，它们的含义与LIKE操作符中一样。单独给出的user\_name（即后面没有给出主机名的情况）相当于'user\_name'@'%'。从MySQL 3.23版本开始，host\_name还允许以n.n.n.n/m.m.m.m形式的IP数字/网络掩码对来给出，其中的n.n.n.n是一个IP地址，m.m.m.m则是一个相应的网络号掩码。

IDENTIFIED BY子句（如果给出的话）用来给指定账户设定口令。不过，这里的口令要求以明文的形式给出，不需要使用PASSWORD()函数——这与通过SET PASSWORD语句来给某个账户设定口令的做法是不同的。如果指定账户已经存在且IDENTIFIED BY子句设定了一个新口令，该账户的老口令就将被替换为新设定的口令；否则，该账户的老口令将保持不变。

REQUIRE子句（如果给出的话）表示指定账户需要使用安全化连接来连接MySQL服务器并要求客户（程序）提供相应的信息。可以在REQUIRE关键字的后面给出以下选项：

- NONE：不要求指定账户必须通过安全化连接来连接MySQL服务器。
- SSL：指定账户必须通过SSL来连接MySQL服务器。
- X509：指定账户必须提供一份合法的X509证书。这个选项对客户（程序）提供的X509证书的内容并没有具体要求，只要它是合法的就行。



• 以下一个或者多个选项，这几个选项对指定账户所建立的安全化连接和有关客户（程序）所提供的X509证书的内容做出了具体的要求：

- CIPHER '*str*': 指定账户必须以字符串'*str*'作为连接期间的加密密钥。
- ISSUER '*str*': 客户（程序）提供的数字证书的签发者必须是'*str*'。
- SUBJECT '*str*': 客户（程序）提供的数字证书的主题必须是'*str*'。

如果需要给出多个上述选项，可以用AND把它们分隔开（但这个AND分隔符是可选的）。这些选项的先后顺序无关紧要。

WITH子句（如果给出的话）对指定账户是否可以把这条GRANT语句授予给它的权限转授给其他用户的情况做出了规定。从MySQL 4.0.2版本开始，这个子句还可以用来设置账户的资源配额。WITH子句的各种选项如下所示（可以同时使用多个选项，它们的先后顺序无关紧要）：

- GRANT OPTION “转授权”权限，即允许这个账户把它自己的权限——包括“转授权”权限（如果这个账户有这个权限的话）在内——转授给其他账户。
- MAX\_CONNECTIONS\_PER\_HOUR *n* 这个账户每小时内最多允许建立*n*个连接；0表示无限制。
- MAX\_QUERIES\_PER\_HOUR *n* 这个账户每小时内最多允许发出*n*次数据库查询；0表示无限制。
- MAX\_UPDATES\_PER\_HOUR *n* 这个账户每小时内最多允许发出*n*次会对数据库里的数据进行修改的查询；0表示无限制。

GRANT语句最早出现于MySQL 3.22.11版本。REQUIRE子句最早出现于MySQL 4.0.0版本；它的NONE选项以及REQUIRE子句各选项之间可选的AND分隔符最早出现于MySQL 4.0.4版本。资源管理设置选项最早出现于MySQL 4.0.2版本。

下面是一些演示GRANT语句各种用法的示例。第11章中还有许多这方面的示例。第12章对使用SSL来建立安全化连接的问题做了比较详细的讨论。

我们先来创建一个名为paul的新账户，并允许他从任何一台主机访问sampdb数据库里的所有数据表（下面两条语句是等价的，因为账户名中缺失的主机名部分相当于“%”）：

```
GRANT ALL ON sampdb.* TO 'paul' IDENTIFIED BY 'secret';
GRANT ALL ON sampdb.* TO 'paul'@'%' IDENTIFIED BY 'secret';
```

接下来，我们再创建一个名为lookonly的新账户，但只允许他从xyz.com域里的任何一台主机以只读方式访问menagerie数据库里的数据表：

```
GRANT SELECT ON menagerie.* TO 'lookonly'@'%.xyz.com'
IDENTIFIED BY 'ragweed';
```

然后，再创建一个名为member\_mgr的新账户，他拥有sampdb数据库里的member数据表上的全部权限（但仅此而已），而且必须从指定主机去连接MySQL服务器：

```
GRANT ALL ON sampdb.member TO 'member_mgr'@'boa.snake.net'
IDENTIFIED BY 'doughnut';
```



下面再来创建一个真正的超级用户，他可以做任何事——包括向其他用户授权在内，但我们限定他只能从本地主机localhost（即MySQL服务器主机本身）去连接MySQL服务器：

```
GRANT ALL ON *.* TO 'superduper'@'localhost' IDENTIFIED BY 'homer'
WITH GRANT OPTION;
```

下面这条语句在menagerie数据库里创建一个匿名用户，他不需要给出口令就能从本地主机localhost连接到MySQL服务器上去：

```
GRANT ALL ON menagerie.* TO ''@'localhost';
```

我们再来为一个远程用户创建一个账户，他必须通过SSL来进行连接，而且必须提供一份合法的X509证书才行：

```
GRANT ALL ON privatedb.*
TO 'paranoid'@'%'.mydom.com IDENTIFIED BY 'keepout'
REQUIRE X509;
```

下面这条GRANT语句所创建的账户每小时内只能进行100次数据库查询，而在这些查询中，只有10次允许是数据修改操作：

```
GRANT ALL ON test.*
TO 'caleb'@'localhost' IDENTIFIED BY 'rosepetal'
WITH MAX_QUERIES_PER_HOUR 100 MAX_UPDATES_PER_HOUR 10;
```

### D.1.23 HANDLER

语法：HANDLER tbl\_name OPEN [AS alias\_name]

```
HANDLER tbl_name READ
[FIRST | NEXT]
[where_clause] [limit_clause]
```

```
HANDLER tbl_name READ index_name
[FIRST | NEXT | PREV | LAST | [< | <= | = | => | >] (expr_list)]
[where_clause] [limit_clause]
```

```
HANDLER tbl_name CLOSE
```

这几条HANDLER语句提供了一个面向MyISAM和InnoDB数据表处理程序的底层接口，它可以绕过MySQL优化器而直接对数据表的内容进行访问。如果你想通过HANDLER接口去访问某个数据表，首先要用HANDLER...OPEN语句去打开它。这个数据表在你发出HANDLER...CLOSE语句（明确地关闭这个数据表）或者结束本次连接（隐含地关闭这个数据表）之前将一直保持在打开状态。在数据表处于打开时，可以用HANDLER...READ语句去访问它的内容。

HANDLER接口没有提供任何针对数据并发修改情况（即同时有多个客户试图修改有关数据）的保护措施。它不对数据表进行锁定，所以用HANDLER...OPEN语句打开的数据表不会阻塞其他客户（程序）对其结构或者内容的修改，而这些改变不一定能反映在你从文件读出来的记录里。

MyISAM数据表上的HANDLER接口最早出现于MySQL 4.0.0版本，InnoDB数据表上的HANDLER接口最早出现于MySQL 4.0.3版本。

#### D.1.24 INSERT

```
语法: INSERT [LOW_PRIORITY | DELAYED] [IGNORE] [INTO]
      tbl_name [(column_list)]
      VALUES (expr [, expr] ...) [, (...)] ...
```

```
INSERT [LOW_PRIORITY | DELAYED] [IGNORE] [INTO]
      tbl_name [(column_list)]
      SELECT ...
```

```
INSERT [LOW_PRIORITY | DELAYED] [IGNORE] [INTO]
      tbl_name SET col_name=expr [, col_name=expr] ...
```

把数据行插入到一个已经存在的数据表`tbl_name`里去并返回实际插入的数据行的个数。在MySQL 3.22.5及以后的版本里，INTO关键字是可选的。

LOW\_PRIORITY选项（如果给出的话）将使INSERT语句被延缓到没有客户（程序）对数据表进行读操作时才得以执行。LOW\_PRIORITY选项最早出现于MySQL 3.22.5版本。

DELAYED选项将把有关数据行放到一个队列里去等待插入，而客户（程序）用不着等待INSERT语句实际完成就能去继续处理下一条SQL语句。这样做的好处是提高了速度，但LAST\_INSERT\_ID()函数却将无法正确地返回AUTO\_INCREMENT数据列的AUTO\_INCREMENT编号值。DELAYED选项最早出现于MySQL 3.22.15版本；但它们目前还只适用于ISAM和MyISAM数据表。

有待插入的数据行有可能导致数据表中现有的惟一化索引里出现重复的键值。如果在INSERT语句里给出了IGNORE选项，导致键值重复的那些数据行就会被丢弃，但插入操作仍将继续进行；如果没有给出IGNORE选项，INSERT语句将报告出错并不再继续插入后续的数据行。IGNORE选项最早出现于MySQL 3.22.10版本。

INSERT语句的第一种形式要求把有待插入的数据行全都写在VALUES()部分里，VALUES()中的每一个`expr`对应着一个有待插入的数据行：如果没有`column_list`部分，每个`expr`中的元素个数就都必须等于数据表的数据列个数；如果有`column_list`部分（它由彼此以逗号分隔的一个或者多个数据列名称构成），每个`expr`中的元素个数就都必须等于`column_list`部分所列举的数据列个数，而没有在`column_list`部分里出现的数据列将被设置为它们的默认值。从MySQL 3.22.5版本开始，INSERT语句的VALUES()部分允许包含多个`expr`，即允许使用一条INSERT语句插入多个数据行。从MySQL 3.23.3版本开始，INSERT语句的`column_list`和VALUES()部分允许同时为空，即允许你往数据表里插入一条各数据列全部是其默认值的数据行，如下所示：

```
INSERT INTO t () VALUES();
```

从4.0.3版本开始，MySQL允许在VALUES()部分用关键字DEFAULT把待插入数据行里的某个数据列明确地设定为它的默认值（你不需要知道这个默认值具体是什么）。

INSERT语句的第二种形式将先执行相应的SELECT语句，然后再把检索到的记录插入到数据表`tbl_name`里去。SELECT语句所选取的数据列的个数必须等于数据表`tbl_name`里的数据列个数（如果INSERT语句没有`column_list`部分的话）或者`column_list`部分所列举的数据列个数（如果INSERT语句有`column_list`部分的话）。如果INSERT语句有`column_list`部分，那些没有在`column_list`里出现的数据列将被设置为它们的默认值。需要特别注意的是，INSERT语句的这种形式要求有关的选取和插入操作分别在不同的数据表上进行，不能把从某个数据表里选取出来的记录再插入到同一个数据表里。

INSERT语句的第三种形式始见于MySQL 3.22.10版本，它将把SET子句生成的数据行插入到数据表里去。SET子句负责把各有关数据列设置为相应的表达式的值，没有出现在SET子句里的数据列将被设置为它们的默认值。

```
INSERT INTO absence (student_id, date) VALUES(14, '1999-11-03'), (34, NOW());
INSERT INTO absence SET student_id = 14, date = '1999-11-03';
INSERT INTO absence SET student_id = 34, date = NOW();
INSERT INTO score (student_id, score, event_id)
    SELECT student_id, 100 AS score, 15 AS event_id FROM student;
```

#### D.1.25 KILL

语法: KILL `thread_id`

终止`thread_id`指定的服务器线程（即终止该线程的执行）。你必须拥有SUPER权限（MySQL 4.0.2及以后的版本）或PROCESS权限（MySQL 4.0.2版本之前）才能终止不属于你本人的线程。KILL语句每次只能终止一个线程，而能够完成同样操作的mysqladmin kill命令则允许在命令行上同时给出多个线程的ID编号。

这条语句最早出现于MySQL 3.22.9版本。

#### D.1.26 LOAD DATA

```
语法: LOAD DATA [LOW_PRIORITY | CONCURRENT] [LOCAL] INFILE 'file_name'
    [IGNORE | REPLACE]
    INTO TABLE tbl_name
    import_options
    [IGNORE n LINES]
    [(column_list)]
```

LOAD DATA语句读出文件`file_name`里的记录并把它们批量加载到数据表`tbl_name`里去。这要比使用一组INSERT语句来完成同样的工作速度快。

LOW\_PRIORITY选项（如果给出的话）将使LOAD DATA语句被延缓到没有客户（程序）对数据表进行读操作时才得以执行。LOW\_PRIORITY选项最早出现于MySQL 3.23.0版本。

CONCURRENT选项只适用于MyISAM数据表，它允许其他客户在你往数据表里加载数据的同时对数据表进行检索（读）操作。CONCURRENT选项最早出现于MySQL 3.23.38版本。

如果LOAD DATA语句不带LOCAL关键字，就将由MySQL服务器在服务器主机上直接读取数据文件。这一操作要求：1）你本人必须具备FILE权限；2）数据文件必须位于默认数据库的数据库目录中或者是可读的。如果LOAD DATA语句带LOCAL关键字，就将由客户（程序）在客户主机上寻找和读取数据文件并把其内容通过网络发送给MySQL服务器，这一操作不要求你必须具备FILE权限。LOCAL机制最早出现于MySQL 3.22.15版本。但从MySQL 3.23.49版本开始，LOCAL机制可以有选择地被禁用或者激活：如果服务器端禁用了LOCAL机制，你就无法在客户端使用这一机制；如果服务器端激活了LOCAL机制、而客户端却默认地禁用了这一机制，就需要由你去明确地激活它。比如说，如果你使用的是mysql客户程序，就可以通过--local-infile选项去激活LOCAL机制。

如果LOAD DATA语句不带LOCAL关键字，MySQL服务器将按以下原则去寻找数据文件：

- 如果'*file\_name*'是一个绝对路径名，服务器就将到该路径指定的地方去读取之。
- 如果'*file\_name*'是一个相对路径名，那就还要看它是否只有一个成分：如果它只有一个成分，服务器就将到当前的默认数据库的数据库目录里去寻找数据文件；如果它包含有多个成分，MySQL服务器将从它自己的数据目录开始去寻找数据文件。

如果LOAD DATA语句带LOCAL关键字，客户（程序）将按以下原则去寻找数据文件：

- 如果'*file\_name*'是一个绝对路径名，客户（程序）就将到该路径指定的地方去读取之。
- 如果'*file\_name*'是一个相对路径名，客户（程序）就将从当前目录开始去寻找数据文件。

对于Windows系统，文件名中的反斜线字符既可以写成斜线字符（/），也可以写成双反斜线字符（\\）。

来自数据文件的数据行有可能导致数据表中现有的惟一化索引里出现重复的键值。如果在LOAD DATA语句里给出了IGNORE关键字，会导致键值重复的那些数据行被丢弃，但数据加载操作仍将继续进行；如果给出了REPLACE关键字，会导致键值重复的那些数据行替换掉数据表中现有的数据行，数据加载工作继续进行；如果这两个关键字都没有给出，LOAD DATA语句将报告出错并不再继续加载后续的数据行。但在LOAD DATA语句使用了LOCAL关键字的情况下，因为MySQL服务器并不能停止文件传输操作，所以LOAD DATA LOCAL语句在IGNORE或REPLACE关键字都没有给出时的默认行为将等价于给出了IGNORE关键字时的情况。

*import\_options*子句负责表明数据的格式；这个子句的可用选项也同样适用于SELECT ... INTO OUTFILE语句中的*export\_options*子句。*import\_options*子句的语法如下所示：

```
[FIELDS
  [TERMINATED BY 'string']
  [[OPTIONALLY] ENCLOSED BY 'char']
  [ESCAPED BY 'char' ] ]
[LINES
  [STARTING BY 'string']
  [TERMINATED BY 'string' ] ]
```

*import\_options*子句中的'*string*'和'*char*'值允许包含以下转义序列所对应的特殊字符：

转义序列	含 义
\0	ASCII 0
\b	退格符
\n	换行符
\r	回车符
\s	空格
\t	制表符
\'	单引号
\"	双引号
\\	反斜线字符

从MySQL 3.22.10版本开始，还可以用十六进制常数来表示任意字符。比如说，“`LINES TERMINATED BY 0x02`”表示数据文件里的各行数据是以Ctrl-B（ASCII 2）字符结尾的。

如果给出了FIELDS关键字，那么至少还要再给出TERMINATED BY、ENCLOSED BY和ESCAPED BY等子句中的一个；但如果给出多个子句，它们的先后顺序可以随意。类似地，如果给出了LINES关键字，那么至少还要再给出STARTING BY或TERMINATED BY子句中的一个；但如果给出多个子句，它们的先后顺序可以随意。如果同时给出了FIELDS和LINES关键字，就必须把FIELDS部分安排在LINES部分的前面。

FIELDS子句各组成部分的用途和用法如下所示：

- **TERMINATED BY 'string'** 表明数据文件中同一行上的各项数据值将以字符串'*string*'作为分隔符。
- **ENCLOSED BY 'char'** 表明数据文件中的数据值将以字符'*char*'（如果给出的话）作为引号符；在把有关数据插入到数据表里之前，MySQL会把这个引号符去掉。OPTIONALLY关键字是否给出并不影响ENCLOSED BY '*char*'子句的效果。对于输出（即SELECT ... INTO OUTFILE）语句，由ENCLOSED BY '*char*'子句所设定的字符将被当做用来封闭各有关字段值的引号符；如果同时给出了OPTIONALLY关键字，则只给来自CHAR和VARCHAR数据列的数据值添加相应的引号符。

如果数据文件中的数据值本身就包含有ENCLOSED BY '*char*'子句所设定的引号符，就必须双写这个引号符或者用ESCAPED BY '*char*'子句所设定的转义序列引导符进行转义。否则，数据值中的引号符就会被解释为字段值的结束标记。对于输出（即SELECT ... INTO OUTFILE）语句，MySQL将自动地在字段值中的引号符前面加上一个转义序列引导符。

- **ESCAPED BY 'char'** 表明数据文件中的数据值将以字符'*char*'作为转义序列引导符。在下面的例子里，假设转义序列引导符是反斜线字符（\）。对于输入（即LOAD DATA）语句，不带引号的转义序列\N（反斜线字符“\”+字母“N”）被解释为NULL；转义序列\0（反斜线字符“\”+ASCII 0）被解释为一个取值为零的字节；其他转义序列都将被解释为去掉转义序列引导符之后剩余的东西（即无特殊意义）——比如说，不管字段值是不是已经用双引号引了起来，其中的转义序列“\”都将被解释为双引号字符“”。

对于输出（即SELECT ... INTO OUTFILE）语句，转义序列引导符（仍以“\”为例）将



把NULL值编码为不带引号的转义序列\N（反斜线字符“\”+字母“N”），而字段值本身所包含的ENCLOSED BY字符和ESCAPED BY字符将像字段分隔符和行分隔符那样被加上一个前导的转义序列引导符。但是，如果ESCAPED BY字符是一个空字符（即给出的是ESCAPED BY ' '子句的时候），则不进行转义处理。如果想把转义序列引导符设定为反斜线字符“\”，就必须双写它（即给出一条ESCAPED BY '\\'子句）。

LINES子句各组成部分的用途和用法如下所示：

- LINES STARTING BY 'string' 表明数据文件中的各行数据将以字符串'string'开始。
- LINES TERMINATED BY 'string' 表明数据文件中的各行数据将以字符串'string'结尾。

如果FIELDS和LINES子句都没有给出，则各特殊功能字符将分别使用如下所示的默认值：

```
FIELDS
    TERMINATED BY '\t'
    ENCLOSED BY ''
    ESCAPED BY '\\'
LINES
    STARTING BY ''
    TERMINATED BY '\n'
```

也就是说，如果没有给出LOAD DATA语句的FIELDS和LINES子句，那么在数据文件里，同一行上的数据将以制表符分隔且不带引号、转义序列引导符将默认为反斜线字符“\”、各行数据将以换行符结尾。

如果FIELDS子句所设定的TERMINATED BY字符和ENCLOSED BY字符都是空字符，就表示数据文件将使用固定宽度的数据行格式，各字段值之间也没有分隔符。数据列值将根据数据列的显示宽度从数据文件里被读出或者被写入——对于输出（即SELECT... INTO OUTFILE）语句——数据文件。比如说，输入（即LOAD DATA）语句将把VARCHAR(15)和MEDIUMINT(5)数据列分别当做一个15个字符宽和一个5个字符宽的字段读入；而输出（即SELECT... INTO OUTFILE）语句将把它们分别写为一个15个字符宽和一个5个字符宽的字段。NULL值将被写为由空格符构成的字符串。

输入数据文件里的NULL值是用不带引号的转义序列\N来表示的。如果FIELDS ENCLOSED BY字符不为空，那么所有的非NULL输入值就都将用给定的引号符引起来，而不带引号的单词NULL也将被解释为一个NULL值。

如果给出了IGNORE *n* LINES子句，则输入数据文件的前*n*行将被丢弃。比如说，如果数据文件的第一行是一个标题栏而你又不想把这个标题栏也放到数据库中的数据表里去，就需要使用一个IGNORE 1 LINES子句，如下所示：

```
LOAD DATA LOCAL INFILE 'mytbl.txt' INTO TABLE mytbl IGNORE 1 LINES;
```

如果LOAD DATA语句没有column\_list部分，MySQL将把输入行里的数据依次赋值给数据表中的每一个数据列；如果有column\_list部分（它由彼此以逗号分隔的一个或者多个数据列名称构成），MySQL将把输入行里的数据依次赋值给每一个给定的数据列。名称没有在column\_list部分里出现的数据列将被设置为它们的默认值。如果输入行提供的数据值的个数少于数据列的

个数, 那些没有得到赋值的数据列就都将被设置为它们的默认值。

需要提醒大家注意的是, 在Windows系统上生成的文本文件其行结束符通常是回车加换行两个字符。如果输入数据文件是一个在Windows系统上生成的、以制表符为字段分隔符的文本文件, 那么用来加载这个数据文件的LOAD DATA语句通常可以使用默认的字段分隔符, 但需要另行设定一个行结束符(“\r”代表一个回车符, “\n”代表一个换行符), 如下所示:

```
LOAD DATA LOCAL INFILE 'mytbl.txt' INTO TABLE mytbl
  LINES TERMINATED BY '\r\n';
```

还有一点需要注意: 在Windows系统上, 有些程序所创建的数据文件是以老式的MS-DOS表示法Ctrl-Z字符作为文件尾标记的。在加载这类数据文件时, MySQL数据库里很可能会出现格式错乱的记录。因此, 最好选用一种不使用Ctrl-Z字符作为文件尾标记的Windows程序来生成数据文件, 或者在加载完数据文件后把格式错乱的记录删除掉。

在CSV (comma-separated values, 逗号分隔) 格式的数据文件里, 字段值之间的分隔符是一个逗号, 并且每个字段都放在一对双引号里。这类数据文件可以用下面这条LOAD DATA语句来加载(假设数据文件的行结束符是换行符):

```
LOAD DATA LOCAL INFILE 'mytbl.txt' INTO TABLE mytbl
  FIELDS TERMINATED BY ',' ENCLOSED BY '"';
```

下面这条语句可以用来加载这样一个数据文件里的记录: 它的字段分隔符是Ctrl-A (ASCII 1) 字符, 行结束符是Ctrl-B (ASCII 2) 字符:

```
LOAD DATA LOCAL INFILE 'mytbl.txt' INTO TABLE mytbl
  FIELDS TERMINATED BY 0x01 LINES TERMINATED BY 0x02;
```

#### D.1.27 LOAD ... FROM MASTER

**语法:** LOAD DATA FROM MASTER

```
LOAD TABLE tbl_name FROM MASTER
```

这些语句是镜像机制中的从服务器向主服务器请求发送数据用的。LOAD DATA FROM MASTER语句要求主服务器把所有数据表的数据都发送过来, 它还会刷新从服务器上的镜像同步控制信息, 这样, 从服务器只需对上一次LOAD ... FROM MASTER操作之后主服务器上又发生的修改进行刷新就能实现与主服务器的同步了。

如果在启动从服务器时还给出了一些--replicate-xxx选项, 这些选项将对LOAD DATA FROM MASTER操作传输哪些数据表这一问题产生影响。LOAD DATA FROM MASTER语句本身也有一些必须满足的先决条件, 这方面的详细讨论请参阅本书第11.7.2节。如果这些条件都能得到满足的话, LOAD DATA FROM MASTER语句就能完成主-从服务器之间的数据同步工作。

LOAD TABLE ... FROM MASTER语句将把指定数据表的一份拷贝从主服务器传输到从服务器。这个语句的主要用途是对镜像机制进行调试。

LOAD TABLE ... FROM MASTER语句最早出现于MySQL 3.23.19版本, LOAD DATA FROM MASTER语句最早出现于MySQL 4.0.0版本。

## D.1.28 LOCK TABLE

语法: LOCK {TABLE | TABLES}  
tbl\_name [AS alias\_name] lock\_type  
[, tbl\_name [AS alias\_name] lock\_type] ...

申请数据锁（即对给定的一个或者多个数据表进行锁定）；如有必要，则等到数据锁全都申请到手为止。数据锁类型`lock_type`必须是下列情况之一：

- **READ** 申请读锁定。这种数据锁将阻塞其他客户（程序）对数据表的写操作，但允许其他客户（程序）对数据表进行读操作。
- **READ LOCAL** 这是READ锁定的一个变体，是专为配合并发插入操作而设计的。它只能用在MyISAM数据表上，并且要求MyISAM数据表不能有任何因记录删除操作而产生的空洞。READ LOCAL允许你明确地锁定一个数据表，但如果它是一个MyISAM数据表且内部没有空洞，则仍允许其他客户（程序）对它进行并发插入操作。（如果数据表内部存在空洞，这种锁定将被视为一个普通的READ锁定。）
- **WRITE** 申请写锁定。这种数据锁将阻塞其他客户（程序）对数据表的任何读、写操作。
- **LOW\_PRIORITY WRITE** 申请低优先级写锁定。如果已经有某个客户（程序）正在对给定数据表进行读操作，则等待那个读操作的完成。如果在等待期间又有其他客户（程序）要求读取给定数据表，则允许其他客户进行读操作。只有当没有客户（程序）对给定数据表进行读操作时，才能获得LOW\_PRIORITY WRITE锁定。

LOCK TABLE语句允许你给待锁定的数据表起一个别名，这样，当在今后的查询命令里需要用到这个数据表时，就可以用它的别名来指称它。（如果需要在同一条查询命令里多次用到同一个数据表，就必须为数据表的每一次引用分别申请一个数据锁，即必须在申请数据锁的时候给有关数据表起一个或者多个别名。）

LOCK TABLE语句会释放你当前持有的全部数据锁。也就是说，如果想锁定多个数据表，就必须用一条LOCK TABLE语句把它们全都锁上。客户（程序）在结束运行的时候会自动释放它持有的全部数据锁。

LOW\_PRIORITY WRITE锁定最早出现于MySQL 3.22.8版本，READ LOCAL锁定最早出现于MySQL 3.23.11版本。

```
LOCK TABLES student READ, score WRITE, event READ;
LOCK TABLE member READ;
LOCK TABLES t AS t1 READ, t AS t2 READ;
```

## D.1.29 OPTIMIZE TABLE

语法: OPTIMIZE { TABLE | TABLES } tbl\_name [, tbl\_name ] ...

DELTETE、REPLACE和UPDATE等语句会使数据表的内部出现碎片和空洞，如果数据表有可变长度的数据行，情况就更是如此。为了消除这些碎片和空洞，OPTIMIZE TABLE语句将进行以下操作动作：

- 对数据表进行碎片整理，消除其中的未使用区域，缩小数据表的尺寸。
- 把因碎片化而散布在存储空间各处的可变长度数据行的内容合并在一起，让各数据行的内容都无间断地存放在同一处。
- 如有必要，刷新索引页面。
- 对数据表的内部统计信息进行更新。

OPTIMIZE TABLE语句只能用在MyISAM和BDB数据表上（它在BDB数据表上的行为等价于ANALYZE TABLE语句）。它要求你必须具备各有关数据表上的SELECT和INSERT权限。

OPTIMIZE TABLE语句在功能和效果上与带--check-only-changed、--quick、--sort-index和--analyze选项的myisamchk客户程序相同。但在使用myisamchk客户程序的时候，必须设法阻止MySQL服务器在你对各有关数据表进行检查的过程中去访问它们。使用OPTIMIZE TABLE语句就不必这么麻烦了，服务器在完成各项检查工作的同时，还会替你阻塞其他客户在某个数据表正在被优化的过程中对它的访问。

OPTIMIZE TABLE语句产生的输出报告的格式与CHECK TABLE语句的情况相同。

OPTIMIZE TABLE语句最早出现于MySQL 3.22.7版本。

### D.1.30 PURGE MASTER LOGS

**语法：** PURGE MASTER LOGS TO 'log\_name'

把服务器上早于给定日志文件log\_name的二进制变更日志全部删除；刷新二进制变更日志的索引文件，使它只列出那些未被删除的日志。当你在镜像机制中的从服务器上运行完SHOW SLAVE STATUS语句之后，往往需要使用这个语句来查知仍在使用的日志文件都有哪些。这条语句要求你必须具备SUPER权限或者PROCESS权限（在MySQL 4.0.2之前的版本里）。

下面这条语句将把二进制变更日志binlog.001到binlog.009（或者更精确地讲，它们当中现仍存在的）全部删除，并使binlog.010成为未被删除的日志文件中的第一个：

```
PURGE MASTER LOGS TO 'binlog.010';
```

PURGE MASTER LOGS 语句最早出现于MySQL 3.23.28版本。

### D.1.31 RENAME TABLE

**语法：** RENAME { TABLE | TABLES } tbl\_name TO new\_tbl\_name [, . . . ]

对一个或者多个数据表重新命名。这条语句与 ALTER TABLE . . . RENAME语句很相似，但RENAME TABLE语句能够同时对多个数据表进行重新命名并能在命名过程中锁定它们。如果需要确保给定数据表在被重新命名的过程中不被其他客户（程序）修改，就应该使用RENAME TABLE语句。

RENAME TABLE语句最早出现于MySQL 3.23.23版本。

### D.1.32 REPAIR TABLE

**语法：** REPAIR { TABLE | TABLES } tbl\_name [, tbl\_name ] . . . [ options ]

这条语句的用途是对受损数据表进行修复。它只能用在MyISAM数据表上，并且要求你必须具备各有关数据表上的SELECT和INSERT权限。这条语句中的`options`部分（如果给出的话）是由一个或者多个下列可选项所组成的清单（彼此之间不需要用逗号分隔开）：

- **EXTENDED** 进行包括重建索引等工作在内的高级修复。它与运行`myisamchk --safe-recover`命令来修复数据表的情况相似，只是数据表的修复工作将由MySQL服务器而不是由一个外部工具程序来进行。
- **QUICK** 只对数据表的索引进行修复，不涉及数据表的数据文件。
- **USE\_FRM** 利用数据表的说明文件（即`.frm`文件）来确定其数据文件的内容需要如何解释，然后再利用数据表的数据文件来重新建立其索引文件。如果你弄丢了数据表的索引文件或者它们损坏得无法修复，这个选项就有用了。

不带任何选项的REPAIR TABLE语句在效果上与执行`myisamchk --recover`命令的情况相同。

REPAIR TABLE语句产生的输出报告的格式与CHECK TABLE语句的情况相同。

REPAIR TABLE语句最早出现于MySQL 3.23.14版本。它的QUICK和USE\_FRM选项分别始于MySQL 3.23.16和MySQL 4.0.2版本。在从MySQL 3.23.14到3.23.25版本里，这个语句每次只允许给出一个选项，而且必须以“TYPE =”的形式给出；在3.23.25之后的版本里，“TYPE =”被弃用，而且允许同时给出多个选项。

### D.1.33 REPLACE

```
语法：REPLACE [LOW_PRIORITY | DELAYED] [INTO]
      tbl_name [(column_list)]
      VALUES (expr [, expr] ...) [, (...)] ...
```

```
REPLACE [LOW_PRIORITY | DELAYED] [INTO]
      tbl_name [(column_list)]
      SELECT ...
```

```
REPLACE [LOW_PRIORITY | DELAYED] [INTO]
      tbl_name SET col_name=expr [, col_name=expr] ...
```

REPLACE语句与INSERT语句很相似，但如果将被插入的数据行会导致数据表里的惟一化索引出现重复键值，MySQL将先删除数据表里原有的数据行，然后再插入新记录；从效果上看，就是新数据行替换掉了原有的老数据行。因此，REPLACE语句的语法里不存在IGNORE选项。详细情况请参见前面内容里的INSERT条目。

如果数据表有多个惟一化索引，就有可能发生一条REPLACE语句删除了多个现有数据行的事情：当新数据行同时与几个惟一化索引中的某个值分别实现匹配时，MySQL会先把所匹配的数据行都删除掉，然后再插入新数据行。

REPLACE语句要求你必须具备给定数据表上的INSERT和DELETE权限。在MySQL 4.0.5之前的版本里，它还要求你必须具备给定数据表上的UPDATE权限。



## D.1.34 RESET

语法: RESET option [, option ] . . .

RESET语句对日志和缓存区信息的影响与FLUSH语句中的情况相同。(事实上, RESET最初就是FLUSH语句的一个子句。)允许用做这条语句的option值的可用选项及其作用如下所示:

- MASTER 删除某个镜像主服务器上现有的各有关二进制变更日志, 重新创建一个新的日志文件并把它编号为001, 再刷新二进制变更日志的索引文件, 使它只列出刚创建的新日志文件。
- QUERY CACHE 清除查询缓存区, 把当前注册在其中的查询命令全部删除掉。(如果只是想消除查询缓存区中的碎片而不是想清除它的全部内容, 应该使用FLUSH QUERY CACHE语句。)
- SLAVE 如果是在一个镜像从服务器上, 这个选项将使MySQL“忘记”镜像同步控制信息(即它当前使用的镜像二进制日志的文件名和这些文件中的读写位置)。

RESET语句要求你必须具备RELOAD权限。

RESET语句最早出现于MySQL 3.23.26版本。它的QUERY CACHE选项最早出现于MySQL 4.0.1版本。在从MySQL 3.23.19到3.23.25版本里, 可以用FLUSH MASTER和FLUSH SLAVE语句来达到RESET MASTER和RESET SLAVE语句的效果。

## D.1.35 RESTORE TABLE

语法: RESTORE { TABLE | TABLES } tbl\_name [, tbl\_name ] . . . FROM 'dir\_name'

用备份目录'dir\_name'里的、由BACKUP TABLE语句创建的文件恢复给定的数据表。'dir\_name'应该是MySQL服务器主机上的某个目录的完整路径名, 数据表的备份文件就保存在这个目录里。有待恢复的数据表必须是尚不存在的。

RESTORE TABLE语句只能用在MyISAM数据表上, 并且要求你必须具备相应的INSERT和FILE权限。恢复操作将只使用各数据表的说明文件和数据文件(即.frm和.MYD文件)来进行, 数据表的各个索引将使用这两个文件里包含的信息重新创建出来。

RESTORE TABLE语句最早出现于MySQL 3.23.25版本。

## D.1.36 REVOKE

语法: REVOKE priv\_type [(column\_list)] [, priv\_type [(column\_list)] ...]  
ON { \*.\* | \* | db\_name.\* | db\_name.tbl\_name | tbl\_name }  
FROM account [, account ] ...

收回指定账户的权限。这条语句中的priv\_type、column\_list和account子句的用途和用法与GRANT语句中的情况一样。ON子句的各个选项也与GRANT语句中的ON子句相同。

REVOKE语句并不会把给定账户从user权限表里删除掉。这意味着你仍可以通过该账户来连接MySQL服务器。如果想彻底删除某个账户, 就必须以手动方式把它从user权限表里删除掉。

(也许今后的MySQL版本会增加自动完成账户删除工作的功能。)

REVOKE语句最早出现于MySQL 3.22.11版本。

下面这条语句将收回superduper@localhost账户的所有权限:

```
REVOKE ALL ON *.* FROM 'superduper'@'localhost';
```

下面这条语句将收回member\_mgr用户在sampdb数据库里的member数据表上进行各种数据修改操作的权限:

```
REVOKE INSERT,DELETE,UPDATE ON sampdb.member
FROM 'member_mgr'@'boa.snake.net';
```

下面这条语句将收回匿名用户从本地主机对menagerie数据库里的pet数据表进行各种操作的权限:

```
REVOKE ALL ON menagerie.pet FROM ''@'localhost';
```

注意, REVOKE ALL ...语句所收回的“全部”权限里不包括GRANT OPTION权限。如果你想收回某个账户的“转授权”权限,就需要明确地给出一条如下所示的REVOKE语句:

```
REVOKE GRANT OPTION ON menagerie.pet FROM ''@'localhost';
```

### D.1.37 ROLLBACK

语法: ROLLBACK

回滚当前事务中的各有关语句所做出的修改,把各有关数据表复原为本次事务开始之前的样子。这条语句只适用于那些支持事务处理机制的数据表类型。(对于那些不支持事务处理机制的数据表类型,SQL语句所做出的修改将立刻反映在有关的数据表里,因而无法得到复原。)

如果事先没有使用BEGIN语句或通过把系统变量AUTOCOMMIT设置为0的办法来禁用自动提交模式,ROLLBACK语句将不起作用。

ROLLBACK语句最早出现于MySQL 3.23.14版本。

### D.1.38 SELECT

语法: SELECT

```
[select_options]
select_list
[
    INTO OUTFILE 'file_name' export_options
  | INTO DUMPFILE 'file_name'
  | INTO @var_name [, @var_name ] ...
]
[FROM tbl_list
[WHERE where_expr]
[GROUP BY {unsigned_integer | col_name | formula} [ASC | DESC] , ...]
```

```
[HAVING where_expr]
[ORDER BY {unsigned_integer | col_name | formula} [ASC | DESC] , ...]
[LIMIT [skip_count,] show_count]
[PROCEDURE procedure_name(arg_list)]
[FOR UPDATE | LOCK IN SHARE MODE] ]
```

SELECT语句通常用来从一个或者多个数据表里检索信息，但因为SELECT语句中除SELECT关键字和select\_list子句之外的成分都是可选的，所以还可以用它对表达式进行求值，如下所示：

```
SELECT 'one plus one =', 1+1;
```

select\_options子句（如果需要给出的话）可以包含一个或者多个下列选项：

- ALL

DISTINCT

DISTINCTROW

这几个选项控制着是否需要返回重复的数据行。ALL表示将返回所有的数据行，它也是这几个选项的默认值。DISTINCT和DISTINCTROW表示将从结果集里剔除重复的数据行。

- HIGH\_PRIORITY

当有客户（程序）正在读取某个数据表的时候，对这个数据表进行写操作的其他语句（比如INSERT和UPDATE）就必须等待这个读操作完成后才能执行。如果你在这时发出了一条不带HIGH\_PRIORITY选项的SELECT语句来检索数据表，MySQL服务器就将把你的这条SELECT语句安排在写操作语句（INSERT和UPDATE等）完成后才执行。可如果你在这条SELECT语句里给出了HIGH\_PRIORITY选项，就能使这条SELECT语句获得高于那些写操作语句的优先级——MySQL服务器（在完成正在进行的读操作之后）将优先执行你的SELECT语句，然后再去执行那些已经等待多时的写操作语句。总之，HIGH\_PRIORITY选项能够提高SELECT语句的优先级，从而使MySQL服务器尽可能早地去执行它。不过，因为这个选项会延缓写语句的执行，所以应该只在那些会很快执行完成并且需要立刻执行的SELECT语句里才使用这个选项。HIGH\_PRIORITY选项最早出现于MySQL 3.22.9版本。

- SQL\_BUFFER\_RESULT

在处理完SELECT语句之后，MySQL还要花时间去把查询结果发送回客户端。在此期间，如果SELECT语句不带SQL\_BUFFER\_RESULT选项，各有关数据表将仍处于锁定状态；而如果SELECT语句带SQL\_BUFFER\_RESULT选项，MySQL服务器就将把查询结果另外存放在一个临时数据表里并解除各有关数据表的锁定状态。换句话说，SQL\_BUFFER\_RESULT选项将使MySQL服务器尽可能早地解除各有关数据表的锁定状态，从而使其他客户（程序）得以尽可能早地开始访问那些数据表。（但使用这个选项会消耗更多的磁盘空间和内存。）SQL\_BUFFER\_RESULT选项最早出现于MySQL 3.23.13版本。

- SQL\_CACHE

SQL\_NO\_CACHE

如果查询缓存区处于DEMAND模式（见SET语句条目中的SQL\_QUERY\_CACHE\_TYPE选项），SQL\_CACHE选项将导致SELECT语句的查询结果被缓存起来。SQL\_NO\_CACHE选项将禁用对查询结果的各种缓存机制。这几个选项最早出现于MySQL 4.0.1版本。

- SQL\_CALC\_FOUND\_ROWS

在默认的情况下，带有LIMIT子句的SELECT语句所返回的数据行计数值将等于实际返回的数据行的个数。如果在这类SELECT语句里使用了SQL\_CALC\_FOUND\_ROWS选项，MySQL服务器就将把SELECT语句在不带LIMIT子句的情况下会返回多少个数据行的计数值也统计出来——如果你想知道这个计数值到底是多少，请在SELECT语句返回后立刻发出一个SELECT FOUND\_ROWS()语句。

- SQL\_SMALL\_RESULT

- SQL\_BIG\_RESULT

这两个关键字反映了对结果集尺寸是小还是大的一个估计，优化器可以根据这一信息更好地处理SELECT语句。SQL\_SMALL\_RESULT和SQL\_BIG\_RESULT选项分别始见于MySQL 3.22.12和MySQL 3.23.0版本。

- STRAIGHT\_JOIN

强制数据表必须按它们在FROM子句中的先后顺序进行关联。如果你认为MySQL优化器做出的不是最佳选择，就可以利用这个选项让SELECT查询按你指定的关联顺序去检索各有关数据表。

*select\_list*子句用来列举SELECT语句将要返回的输出列。多个输出列之间要用逗号彼此分隔开。输出列可以是数据表中的数据列，也可以是MySQL表达式。还可以利用AS *alias\_name*语法给输出列起一个别名。输出列的别名将成为输出报告中的列标题，并可以用在GROUP BY、ORDER BY和HAVING等子句里以实现相应的处理。注意：在WHERE子句里不允许使用别名。

可以在*select\_list*子句里使用几种特殊的记号：星号（\*）代表“FROM子句所给定的数据表里的全体数据列”；而*tbl\_name.\**则代表“数据表*tbl\_name*里的全体数据列”。

SELECT语句的查询结果可以利用一条INTO OUTFILE '*file\_name*'子句写到文件*file\_name*里去。*export\_options*子句的语法与LOAD DATA语句的*import\_options*子句相同，详细情况请参阅前面内容里的LOAD DATA条目。

类似于INTO OUTFILE '*file\_name*'子句，INTO DUMPFILE '*file\_name*'子句也可以把查询结果写到指定的文件里去，但它写出来的文件只有一行，而且对输出内容不做任何解释——即把查询结果整个地当做一项数据而不识别其中的分隔符、引号、结束标记等。这个选项特别适合用来把BLOB数据（比如一幅图像或者其他二进制数据等）写到一个文件里去。INTO DUMPFILE子句最早出现于MySQL 3.23.5版本。

对于INTO OUTFILE和INTO DUMPFILE子句中的文件名，MySQL将根据它自己用来解释LOAD DATA语句读取非LOCAL文件时的规则做出解释。你必须具备FILE权限；输出文件必须尚不存在；输出文件将由MySQL服务器创建在服务器主机上；该文件的属主将被设置为用来运行MySQL服务器的那个账户。

从MySQL 4.1版本开始，如果SELECT语句的查询结果只有一个数据行，我们还可以把它保

存到一组由用户定义的变量里去——每个输出列对应一个变量，它们彼此之间要用逗号分隔开。注意，这类变量的名字必须以字符“@”开头（即@var\_nam形式）。

FROM子句用来列举将在其中对数据行进行检索的数据表。MySQL支持在SELECT语句里使用以下关联类型：

```
tbl_list:
    tbl_name
    tbl_list, tbl_name
    tbl_list [CROSS] JOIN tbl_name
    tbl_list INNER JOIN tbl_name ON conditional_expr
    tbl_list INNER JOIN tbl_name USING (column_list)
    tbl_list STRAIGHT_JOIN tbl_name
    tbl_list LEFT [OUTER] JOIN tbl_name ON conditional_expr
    tbl_list LEFT [OUTER] JOIN tbl_name USING (column_list)
    tbl_list NATURAL [LEFT [OUTER]] JOIN tbl_name
    { OJ tbl_list LEFT OUTER JOIN tbl_name ON conditional_expr }
    tbl_list RIGHT [OUTER] JOIN tbl_name ON conditional_expr
    tbl_list RIGHT [OUTER] JOIN tbl_name USING (column_list)
    tbl_list NATURAL [RIGHT [OUTER]] JOIN tbl_name
    (tbl_list)
```

每个数据表名都可以伴随有一个别名或者索引提示。也就是说，在SELECT语句里指称一个数据表的完整语法如下所示：

```
tbl_name
    [[AS] alias_name]
    [USE INDEX (index_list) | IGNORE INDEX (index_list)]
```

如果需要在FROM子句里给数据表起一个别名，可以使用tbl\_name alias\_name或者tbl\_name AS alias\_name语法。别名机制使我们能够在查询命令中的其他地方利用数据表的别名来指称数据表里的数据列。

USE INDEX或IGNORE INDEX子句最早出现于MySQL 3.23.12版本，它们的作用是向MySQL优化器提供一些提示：如果你认为MySQL优化器在数据表关联检索操作中选用的索引不是最佳选择，就可以利用这两个选项来给它点提示（USE KEY和IGNORE KEY分别是USE INDEX和IGNORE INDEX的同义词）。index\_list由一个或者多个以逗号分隔的索引名构成，每个索引名必须对应着数据表中的某个索引或者是关键字PRIMARY（代表数据表的PRIMARY KEY）。

各种关联类型将按以下描述从给定数据表里选取数据行。实际返回给客户（程序）的数据行还将受到WHERE、HAVING或LIMIT子句的限制。

- 如果FROM子句只列举了一个数据表，SELECT将从该数据表检索数据行。
- 如果FROM子句以逗号为分隔列举了多个数据表，SELECT将返回这些数据表里的数据行的全排列组合。使用JOIN或CROSS JOIN与使用逗号等价。STRAIGHT\_JOIN与此类似，但强制MySQL优化器必须按各有关数据表在FROM子句中的先后顺序来进行关联。如果你认为MySQL优化器做出的不是最佳选择，就可以用这个选项。
- INNER JOIN与逗号操作符的含义相同，但还需要像LEFT JOIN那样用一个ON或USING()



子句来约束数据表之间的匹配情况。(注意：在MySQL 3.23.17之前的版本里，INNER JOIN与逗号操作符完全一致且不允许使用ON或USING()子句。)

- 在从有关数据表检索数据行时，LEFT JOIN将强制性地为左数据表里的每一个数据行生成一个数据行，哪怕右数据表里没有与之匹配的数据行也是如此。当没有匹配的时候，来自右数据表的数据列将被返回为NULL值。数据行是否匹配要由ON *conditional\_expr*或USING (*column\_list*)子句来决定。

ON子句中的条件表达式*conditional\_expr*与WHERE子句中的条件表达式格式相同。USING子句中的数据列清单*column\_list*由一个或者多个以逗号分隔的数据列名称构成，这些数据列必须同时出现在关联操作所涉及的左、右两个数据表里。LEFT OUTER JOIN是LEFT JOIN的同义词。{OJ...}语法与此类似——这个语法是为了使MySQL与ODBC标准保持兼容而引入的。(注意：{OJ...}语法中的花括号不是元字符，它们必须原样出现在相应的SELECT语句中。)

- NATURAL LEFT JOIN相当于LEFT JOIN USING (*column\_list*)，其中*column\_list*必须把同时出现在关联操作所涉及的左、右两个数据表里的数据列全都列举出来。
- RIGHT JOIN类型与相应的LEFT JOIN类型相似，但前者把数据表的角色（即关联操作所涉及的左、右两个数据表）给颠倒了过来。RIGHT JOIN子句最早出现于MySQL 3.23.25版本。

MySQL将根据WHERE子句所给出的条件表达式从FROM子句所列举的数据表里选取数据行。(注意：数据列别名是不允许用在WHERE子句里的。)不满足条件表达式的数据行将不会出现在SELECT语句的查询结果里。结果集还要受到HAVING和LIMIT子句的进一步限制。

GROUP BY *column\_list*子句将根据*column\_list*所给定的数据列对结果集里的数据行进行归组。当在*select\_list*子句里使用了COUNT ()或MAX()等统计类函数时，就往往需要使用GROUP BY子句来对结果集里的数据行进行归组。在GROUP BY *column\_list*子句的*column\_list*部分，可以使用数据列的名称、数据列的别名或者数据列在*select\_list*子句里的位置序号（从1开始）来指称各有关数据列。从MySQL 3.23.2版本开始，还可以在GROUP BY子句里使用表达式，即根据表达式的计算结果来对结果集里的数据行进行归组。

在MySQL里，GROUP BY子句不仅会对结果集里的数据行进行归组，还会对它们进行排序。从MySQL 3.23.47版本开始，可以通过在GROUP BY子句中的数据列标识符（名称、别名或序号）的后面加上ASC和DESC关键字的办法来明确地指定排序顺序（从而影响它们的输出顺序），但GROUP BY子句的排序效果会受到ORDER BY子句（如果给出的话）的覆盖。

MySQL将根据HAVING子句所给出的次要条件表达式对那些已经满足WHERE子句所给出的主要条件表达式的数据行做进一步的筛选。不满足HAVING条件的数据行将不会出现在SELECT语句的查询结果里。HAVING子句非常适合用来给出因带有统计函数而无法用在WHERE子句里的条件表达式。但是，如果某个条件表达式在WHERE子句和HAVING子句中都是合法的，就应该把它放到WHERE子句里——因为只有WHERE子句里的条件表达式才会得到MySQL优化器的分析和优化。

ORDER BY子句的用途是让MySQL对结果集里的数据行按指定方式进行排序。类似于

GROUP BY子句中的做法，在ORDER BY子句里，也可以使用数据列的名称、数据列的别名或者数据列在select\_list子句里的位置序号（从1开始）来给出各有关数据列。在默认的情况下，输出列将按升序顺序进行排序。如果想明确地设定某个数据列的排序顺序，就需要在数据列的标识符（名称、别名或序号）的后面加上ASC（升序）或DESC（降序）关键字。从MySQL 3.23.2版本开始，还可以在ORDER BY子句里使用表达式，即根据表达式的计算结果来对结果集里的数据行进行排序。比如说，ORDER BY RAND()将以随机顺序返回各有关数据行。但与GROUP BY子句不同的是，ORDER BY子句里的表达式不允许使用统计类函数。

LIMIT子句的用途是从结果集里进一步选取它的某个组成部分。这个子句可以带一个或者两个参数（这些参数必须是整数常数）。LIMIT *n*将返回结果集里的前*n*个数据行，LIMIT *m*, *n*则将先跳过结果集里的前*m*个数据行、再返回随后的*n*个数据行。带两个参数的LIMIT子句还允许你通过LIMIT *m*, -1格式来先跳过结果集里的前*m*个数据行、再返回所有剩余的数据行（不管它们有多少）。

PROCEDURE子句给出的是一个过程的名字。如果SELECT语句里有PROCEDURE子句，MySQL服务器就会在把结果集发送回客户（程序）之前先把结果集发送到这个过程去进行一些处理。PROCEDURE子句的参数表arg\_list可以为空，也可以是一个以逗号分隔的参数清单，这些参数将被传递到指定的过程里去。从MySQL 3.23版本开始，可以通过PROCEDURE ANALYSE()子句来获得SELECT语句所选取的各数据列里的数据值的统计性信息。

FOR UPDATE和LOCK IN SHARE MODE子句将锁定SELECT语句所选取的数据行，直到当前事务被提交或者被回滚为止。这在多语句事务里非常有用。在一个支持页面级或数据行级锁定机制的数据表（如BDB或InnoDB数据表）上使用FOR UPDATE子句将在SELECT语句所选取的数据行上施加一个独占性的写锁定。使用LOCK IN SHARE MODE子句将在SELECT语句所选取的数据行上施加一个读锁定，即允许其他客户（程序）去读这些数据行、但不允许它们去修改这些数据行。这两个锁定子句最早出现于MySQL 3.23.35版本。

下面这些语句演示了SELECT语句的一些用法。本书的第1章和第3章里还有更多的示例。

选取数据表（以president数据表为例）的全部内容：

```
SELECT * FROM president;
```

选取president数据表的全部内容，但要根据总统们的姓名进行排序：

```
SELECT * FROM president ORDER BY last_name, first_name;
```

从president数据表里把出生日期是或者晚于'1900-01-01'的总统查出来：

```
SELECT * FROM president WHERE birth >= '1900-01-01';
```

从president数据表里把出生日期是或者晚于'1900-01-01'的总统查出来，但要按出生日期进行排序：

```
SELECT * FROM president WHERE birth >= '1900-01-01' ORDER BY birth;
```

查看member数据表里的数据行涉及到哪些（美国的）州：

```
SELECT DISTINCT state FROM member;
```

选取member数据表里的数据行并把它们写到一个文件里去，各数据列以逗号分隔：

```
SELECT * INTO OUTFILE '/tmp/member.txt'
    FIELDS TERMINATED BY ',' FROM member;
```

从score数据表里把某次考试前5名学生的记录选取出来：

```
SELECT * FROM score WHERE event_id = 9 ORDER BY score DESC LIMIT 5;
```

子查询支持

MySQL 4.1增加了子查询支持机制，即允许一条SELECT语句嵌套在另一条SELECT语句中。

下面是这种子查询机制的几种用法：

- 把子查询机制用在一个比较操作里，此时，子查询应该仅返回一个值。

例如，下面这个查询将把birth值最早的president记录给找出来：

```
SELECT * FROM president
WHERE birth = (SELECT MIN(birth) FROM president);
```

下面这个查询将从score数据表里把某次考试成绩高于平均分的有关记录给找出来：

```
SELECT * FROM score WHERE event_id = 5
AND score > (SELECT AVG(score) FROM score WHERE event_id = 5);
```

- 把子查询机制用在EXISTS或NOT EXISTS子句里。此时，因为内层的SELECT语句可能会用到外层SELECT语句中的数据列，所以通常应该在各有关数据列名称的前面加上它们所在的数据表的名称，以消除二义性。

下面这个查询将把至少缺勤过一次考试的学生给找出来：

```
SELECT student_id, name FROM student WHERE EXISTS
    (SELECT * FROM absence WHERE absence.student_id = student.student_id);
```

下面这个查询将把没有缺勤过考试的学生给找出来：

```
SELECT student_id, name FROM student WHERE NOT EXISTS
    (SELECT * FROM absence WHERE absence.student_id = student.student_id);
```

注意，上面两个语句的子查询部分使用了SELECT \*，这是因为进行内层查询的目的是为了产生一个真或假值而不是某个特定的数据列值。

- 把子查询机制用在IN或NOT IN子句里，此时，子查询的结果集应该有且仅有一个输出列。上面两个EXISTS或NOT EXISTS查询可以用IN或NOT IN子句改写为如下所示的样子：

```
SELECT student_id, name FROM student
WHERE student_id IN (SELECT student_id FROM absence);

SELECT student_id, name FROM student
WHERE student_id NOT IN (SELECT student_id FROM absence);
```

#### D.1.39 SET

语法：SET [ OPTION ] option\_setting [, option\_setting ] . . .

SET语句用来对MySQL数据库系统的各种选项、用户定义变量、全局级或会话级变量以及事务隔离级别进行赋值或设置。SET TRANSACTION ISOLATION LEVEL语句将单独作为一个条目另行说明，而用户定义变量将在第D.2节里进行讨论。

在SET语句里使用OPTION关键字的做法现在已经有点过时，今后的MySQL版本很可能彻底淘汰它。

*option\_setting*部分的赋值语法是`name op val`，其中，*name*是将获得赋值的SQL选项或变量的名字，*op*是赋值操作符，而*val*就是将被赋值给有关选项或变量的值。MySQL 3.23.6之前的版本只允许使用等号(=)作为SET语句中的赋值操作符；在MySQL 3.23.6及以后的版本里，“=”和“:=”都可以用做赋值操作符。

SET语句可以用来对某些SQL选项进行赋值，如下所示：

```
SET SQL_LOG_BIN = 1;
SET AUTOCOMMIT = 0;
```

SET语句也可以用来对用户定义的变量进行赋值，如下所示：

```
SET @day = CURDATE(), @time = CURTIME();
```

此外，从MySQL 4.0.3版本开始，MySQL服务器还支持几个动态系统变量，即允许在服务器仍在运行时对这些变量进行修改，这类修改也要靠SET语句来完成。（在MySQL 4.0.3之前的版本里，MySQL系统变量只能在MySQL服务器启动时设置。如果修改了系统变量，就必须重新启动MySQL服务器才能使之生效。）MySQL提供的动态系统变量有两类：全局级变量的有效范围是整个服务器，它们对所有的客户（程序）都有影响；会话级变量（也叫做本地变量）的有效范围只局限于某个特定的客户连接。对于那些同时存在于系统、会话两个级别的变量，每个新建的客户连接都会根据相应的全局级变量的值得到一个初始的会话级变量设置。任何一个客户（程序）都可以修改它自己的会话级变量，但只有那些具备SUPER权限的用户才能修改全局级变量。

全局级变量可以用下面两条语句中的任何一条修改（以`table_type`为例）：

```
SET GLOBAL table_type = InnoDB;
SET @@GLOBAL.table_type = InnoDB;
```

会话级变量则需要用下面两条语句中的任何一条修改：

```
SET SESSION table_type = InnoDB;
SET @@SESSION.table_type = InnoDB;
```

还可以把LOCAL用做SESSION的同义词，如下所示：

```
SET LOCAL table_type = InnoDB;
SET @@LOCAL.table_type = InnoDB;
```

如果在SET语句里没有给出GLOBAL、SESSION或LOCAL关键字，SET语句将默认地修改会话级变量：

```
SET table_type = InnoDB;
SET @@table_type = InnoDB;
```



如果想查看系统变量的设置情况，请使用SHOW VARIABLES语句。本附录后面内容里的SHOW VARIABLES条目对允许用SET语句进行设置的各种变量以及它们是否允许动态地加以修改做了说明。

下面是允许利用SET语句加以修改的SQL选项。有些选项要求必须具备SUPER权限（在MySQL 4.0.2之前的版本里，需要具备PROCESS权限）才能修改。

- **AUTOCOMMIT = { 0 | 1 }**

禁用/激活自动提交模式，以进行事务处理。把这个选项设置为0将禁用自动提交模式，后续语句将作为一个事务被提交（通过发出一条COMMIT语句或者把AUTOCOMMIT选项重新设置为1）。如果尚未被提交，构成这次事务的各个语句对数据库所做出的修改还可以用ROLLBACK语句撤销。把AUTOCOMMIT选项重新设置为1将再次激活自动提交模式（并提交所有尚未被提交的事务）。在自动提交模式处于激活状态期间，SQL语句对数据库做出的修改将在语句执行完毕后立刻生效——实际上，这相当于每条语句本身就是一个事务。

- **CHARACTER SET [=] { charset | DEFAULT }**

设定客户（程序）所使用的字符集。此后，这个客户所发送和接收的字符串都将用该字符集来进行解释。这个选项目前只有cp1251\_koi8这一个字符集名称可供设置。把这个选项设置为DEFAULT将恢复使用当前的默认字符集。从MySQL 4.0.3版本开始，设置这个选项时可以省略等号（=）；在MySQL 4.0.3之前的版本里，这个等号不允许省略。

- **FOREIGN\_KEY\_CHECKS = { 0 | 1 }**

把这个选项设置为0或1将禁用或者激活MySQL的外键检查功能。这个查询的默认设置是激活外键检查功能。某些场合需要禁用MySQL的外键检查功能。比如说，当用一个转储文件来恢复数据表的时候，各有关数据表的创建和加载顺序往往与外键关系所要求的顺序不一致，所以在开始进行数据加载工作之前最好先把MySQL的外键检查功能关闭掉，等数据加载工作完成之后再重新激活之。这个选项最早出现于MySQL 3.23.52版本。它只能用于InnoDB数据表，对其他类型的数据表不起作用。

- **INSERT\_ID = *n***

使下一条INSERT语句往数据表里插入的AUTO\_INCREMENT值等于*n*。变更日志的处理工作经常会用到这个选项。

- **LAST\_INSERT\_ID = *n***

使下一次LAST\_INSERT\_ID()函数调用的返回值等于*n*。变更日志的处理工作经常会用到这个选项。

- **PASSWORD [ FOR account ] = PASSWORD ( 'pass\_val' )**

如果不带FOR子句，这个选项将把当前账户的口令设置为'pass\_val'。如果带FOR子句，这个选项将把指定账户的口令设置为'pass\_val'。如果想修改其他账户的口令，必须具有允许你对mysql数据库进行相应修改的权限。这个选项中的account参数要求以'user\_name'@'host\_name'的格式给出，user\_name和host\_name的语法与GRANT语句中的情况相同，详细讨论请参见本附录前面内容里的GRANT条目。



```

SET PASSWORD = PASSWORD('secret');
SET PASSWORD FOR 'paul' = PASSWORD('secret');
SET PASSWORD FOR 'paul'@'localhost' = PASSWORD('secret');
SET PASSWORD FOR 'bill'@'%.bigcorp.com' = PASSWORD('old-sneep');

```

• **SQL\_AUTO\_IS\_NULL = { 0 | 1 }**

如果这个选项被设置为1, 我们就可以用WHERE *col\_name* IS NULL形式的WHERE子句来查知最近一次生成的AUTO\_INCREMENT编号值 (*col\_name* 就是那个AUTO\_INCREMENT数据列的名字)。有很多ODBC程序, 比如微软公司的Access, 都需要用到这一功能。这个选项的默认设置值是1。这个选项最早出现于MySQL 3.23.5版本。

• **SQL\_BIG\_SELECTS = { 0 | 1 }**

这个选项需要与MAX\_JOIN\_SIZE选项配合使用:

- 如果SQL\_BIG\_SELECTS选项被设置为1 (这是它的默认设置), 那么, 不管查询命令将会返回的结果集的尺寸有多大, MySQL服务器都将毫无怨言地执行它。
- 如果SQL\_BIG\_SELECTS选项被设置为0, 那么, 如果某条查询命令将会返回的结果集尺寸过大, MySQL服务器就不会执行它。此时, MySQL在对多个数据表进行关联检索的时候将受到MAX\_JOIN\_SIZE值的约束: 服务器先估算出需要对多少种数据行组合情况进行检查, 如果这个估算值大于MAX\_JOIN\_SIZE选项的设置值, 服务器就将返回一条出错信息而不是去执行这个查询。

如果把MAX\_JOIN\_SIZE选项设置为不是DEFAULT的其他值, SQL\_BIG\_SELECTS选项就将被自动设置为0。

• **SQL\_BIG\_TABLES = { 0 | 1 }**

BIG\_TABLES = { 0 | 1 }

如果这个选项被设置为1, MySQL内部使用的所有临时数据表就都将被存储到磁盘上而不是驻留在内存里。这会降低MySQL服务器的性能, 但那些会生成大量临时数据表的SELECT语句却不会因产生“table full”(数据表满)错误而无法完成了。这个选项的默认值是0 (把临时数据表存储在内存里)。对于MySQL 3.23及以后的版本, 通常用不着对这个选项进行设置。MySQL 4及以后的版本推荐使用BIG\_TABLES来作为这个选项的名字。

• **SQL\_BUFFER\_RESULTS = *n***

如果这个选项被设置为1, MySQL将使用一些临时数据表来存放SELECT语句的查询结果并会尽可能早地解除各有关数据表的锁定状态, 从而使其他客户(程序)得以尽可能早地开始访问那些数据表(但使用这个选项会消耗更多的磁盘空间和内存)。这个选项最早出现于MySQL 3.23.13版本。

• **SQL\_LOG\_BIN = { 0 | 1 }**

把这个选项设置为1将激活当前客户(程序)上的二进制变更日志机制。把这个选项设置为0关闭该功能。客户(程序)必须具备SUPER权限才能使这条语句发生作用。这个选项最早出现于MySQL 3.23.16版本。

• **SQL\_LOG\_OFF = { 0 | 1 }**

如果这个选项被设置为1, 当前客户(程序)的查询将不会被记录到常规日志文件里去。

如果这个选项被设置为0，则将激活针对这个客户的日志功能。客户（程序）必须具备SUPER权限才能使这条语句发生作用。

- **SQL\_LOG\_UPDATE = { 0 | 1 }**

如果这个选项被设置为1，当前客户（程序）的查询将被记录到服务器的变更日志文件里去。如果这个选项被设置为0，则禁用此项功能。客户（程序）必须具备SUPER权限才能使这条语句发生作用。这个选项最早出现于MySQL 3.22.5版本。

- **SQL\_LOW\_PRIORITY\_UPDATES = { 0 | 1 }**

**LOW\_PRIORITY\_UPDATES = { 0 | 1 }**

如果这个选项被设置为1，对数据表内容进行修改的语句（DELETE、INSERT、REPLACE、UPDATE等）将被延缓到没有SELECT语句在读取或者等待读取该数据表时才得以执行。如果服务器在某个SELECT语句正在执行的时候又接收到了一条SELECT语句，新接收到的SELECT语句将在前一条SELECT语句执行完毕后立刻开始执行而不是等那些低优先级的数据表修改语句执行完毕后才开始执行。这个选项最早出现于MySQL 3.22.5版本。MySQL 4及以后的版本推荐使用LOW\_PRIORITY\_UPDATES来作为这个选项的名字。

- **SQL\_MAX\_JOIN\_SIZE = { n | DEFAULT }**

**MAX\_JOIN\_SIZE = { n | DEFAULT }**

这个选项要与SQL\_BIG\_SELECT选项配合使用，详细情况请参见SQL\_BIG\_SELECT条目。MySQL 4及以后的版本推荐使用MAX\_JOIN\_SIZE来作为这个选项的名字。

- **SQL\_QUERY\_CACHE\_TYPE = { 0 | 1 | 2 | OFF | ON | DEMAND }**

**QUERY\_CACHE\_TYPE = { 0 | 1 | 2 | OFF | ON | DEMAND }**

为当前客户（程序）设置查询缓存区的工作模式，如下表所示。

模 式	含 义
0、OFF	不进行缓存
1、ON	进行缓存，但以SELECT SQL_NO_CACHE开头的查询命令不包括在内
2、DEMAND	只对以SELECT SQL_CACHE开头的查询命令进行缓存

这个选项最早出现于MySQL 4.0.1版本；推荐使用QUERY\_CACHE\_TYPE来作为这个选项的名字。

- **SQL\_QUOTE\_SHOW\_CREATE = { 0 | 1 }**

这个选项控制着SHOW CREATE TABLE语句的输出报告里的数据表、数据列和索引的名字是否需要用反单引号（`）括起来。这个选项的默认设置是1（使用反单引号）。如果想把SHOW CREATE TABLE语句输出的CREATE TABLE语句拿到别的数据库服务器上去使用，或者如果你的MySQL服务器的版本低于3.23.6（老版本的MySQL服务器不支持反单引号的这种用法），就需要把这个选项设置为0以关闭该功能。不过，如果真的关闭了这项功能，就一定要保证数据表、数据列和索引的名字没有使用MySQL保留字或者包含特殊

字符。这个选项最早出现于MySQL 3.23.26版本。

- `SQL_SAFE_UPDATES = { 0 | 1 }`

如果这个选项被设置为1, MySQL将只允许两种情况下的UPDATE和DELETE语句执行: 1) 将被修改的记录是通过索引键值而确定的; 2) 使用了LIMIT子句。这个选项最早出现于MySQL 3.22.32版本。

- `SQL_SELECT_LIMIT = { n | DEFAULT }`

这个选项的设置值决定了SELECT语句能够返回的数据行的最大个数。但如果在SELECT语句里给出了一条LIMIT子句, 则LIMIT子句中的设置情况将优先于这个选项。这个选项的默认设置是“无限制”。如果把这个选项设置为DEFAULT, 则将恢复其默认设置(如果你曾改动过这个选项的话)。

- `SQL_SLAVE_SKIP_COUNTER = n`

让镜像机制中的从服务器跳过来自主服务器的 $n$ 个事件。从服务器线程必须尚未运行或者已经停止。这个选项最早出现于MySQL 3.23.33版本。从MySQL 4.0.3版本开始, 必须通过SET GLOBAL语句而不是SET语句来修改这个选项。

- `SQL_WARNINGS = { 0 | 1 }`

如果这个选项被设置为1, 那么, 即使是只插入一个新数据行的INSERT语句, MySQL也会报告它的警告计数值——如果新插入的数据行会导致数据表中出现重复记录项的话。这个选项的默认设置是0, 即只对插入多个数据行的INSERT语句实施这种出错报告机制。这个选项最早出现于MySQL 3.22.11版本。

- `TIMESTAMP = { timestamp_value | DEFAULT }`

设定一个TIMESTAMP值。变更日志的处理工作经常会用到这个选项。

- `UNIQUE_CHECKS = { 0 | 1 }`

MySQL允许数据表有多个惟一化索引, 这些惟一化索引在各种数据库操作中的作用有主次之分(数据表的主键通常是最主要的惟一化索引)。把这个选项设置为0或1将禁用或者激活MySQL对InnoDB数据表中各个次要的惟一化索引的惟一性检查机制。这个选项最早出现于MySQL 3.23.52版本。

#### D.1.40 SET TRANSACTION ISOLATION LEVEL

语法: `SET [ GLOBAL | SESSION ] TRANSACTION ISOLATION LEVEL level`

如果给出了GLOBAL或SESSION关键字, 这条语句将对全局级(作用于整个服务器)或者会话级(只作用于某个具体的客户)事务隔离级别进行设定; 如果GLOBAL和SESSION关键字都没有给出, 则将为当前会话中的下一次事务设定其事务隔离级别。要想设置全局级事务隔离级别, 必须具备SUPER权限(在MySQL 4.0.2之前的版本里, 必须具备PROCESS权限)。全局级事务隔离级别的改变将影响到客户(程序)在此后新建的所有连接, 但此前已经连接上MySQL服务器的客户(程序)不受影响。

事务隔离级别参数`level`的可取值应该是以下之一: READ UNCOMMITTED、READ

COMMITTED、REPEATABLE READ或者SERIALIZEABLE。

这条语句最早出现于MySQL 3.23.36版本。但它只在MySQL 3.23.50版本之后才对InnoDB数据表处理程序的工作情况有影响；在此之前，InnoDB处理程序总是操作在REPEATABLE READ模式之下。在MySQL 4.0.5版本之前，InnoDB所允许的其他可取值只有SERIALIZEABLE——如果把事务隔离级别设置为其他值，MySQL将使用REPEATABLE READ。从MySQL 4.0.5版本开始，InnoDB数据表才开始支持使用READ COMMITTED和READ UNCOMMITTED事务隔离级别。

BDB数据表处理程序不受这个选项的影响——因为BDB处理程序总是运行在SERIALIZEABLE模式之下。

#### D.1.41 SHOW

语法：SHOW BINLOG EVENTS [IN 'file\_name'] [FROM n]  
 [LIMIT [skip\_count,] show\_count]  
 SHOW CHARACTER SET  
 SHOW COLUMN TYPES  
 SHOW [FULL] COLUMNS FROM tbl\_name [FROM db\_name] [LIKE 'pattern']  
 SHOW CREATE DATABASE db\_name  
 SHOW CREATE TABLE tbl\_name  
 SHOW DATABASES [LIKE 'pattern']  
 SHOW GRANTS FOR account  
 SHOW INDEX FROM tbl\_name [FROM db\_name]  
 SHOW INNODB STATUS  
 SHOW LOGS  
 SHOW MASTER LOGS  
 SHOW MASTER STATUS  
 SHOW PRIVILEGES  
 SHOW [FULL] PROCESSLIST  
 SHOW SLAVE HOSTS  
 SHOW SLAVE STATUS  
 SHOW STATUS [LIKE 'pattern']  
 SHOW TABLE STATUS [FROM db\_name] [LIKE 'pattern']  
 SHOW TABLE TYPES  
 SHOW [OPEN] TABLES [FROM db\_name] [LIKE 'pattern']  
 SHOW [GLOBAL | SESSION] VARIABLES [LIKE 'pattern']

SHOW语句的各种形式使你能够查知关于数据库、数据表、数据列、索引以及服务器运行状态的各种信息。有几种SHOW语句允许使用一个可选的FROM *db\_name*子句，这个子句的作用是告诉MySQL所想了解的是关于哪一个数据库的信息；如果没有给出这个子句，MySQL就会把关于当前默认数据库的信息报告给你（从MySQL 4.0版本开始，这些语句中的FROM关键字还可以写成它的同义词IN）。

此外，有几种SHOW语句允许使用一个可选的LIKE '*pattern*'子句，这个子句的作用是让

SHOW 语句只把那些名称与给定模式相匹配的选项或者变量的值显示给你。'pattern'将被解释为一个SQL匹配模式且允许包含“%”或“\_”通配符。

#### D.1.42 SHOW BINLOG EVENTS

在镜像机制中的主服务器上发出这条语句将显示记录在二进制变更日志里的事件。它的输出报告由以下几个输出列组成：

- Log\_name 二进制日志文件的名称。
- Pos 事件在日志文件中的记录位置。
- Event\_type 事件的类型，比如说，可执行SQL语句的类型是Query。
- Server\_id 负责记录这个事件的服务器的ID编号。
- Orig\_log\_pos 这个事件在主服务器上的原始日志文件里的位置。
- Info 事件信息，比如Query事件的语句文本。

SHOW BINLOG EVENTS语句最早出现于MySQL 4.0版本。

#### D.1.43 SHOW CHARACTER SET

显示一份MySQL服务器当前支持使用的字符集名单。

这条语句的输出报告目前只能提供每种字符集的以下信息，但未来的MySQL版本可能会增加一些新内容。

- Name 字符集的名字。
- Id 该字符集的MySQL内部编号。
- strx\_maxlen 在对以给定字符集写出的数据值进行排序时，MySQL必须为字符串的内部转换操作而分配一些内存。本输出列里的数据就是与给定字符集相关联的内存开销因素。
- mb\_maxlen 给定字符集里“最宽的”字符的长度，以字节为计量单位。对于多字节字符集，这个值是一个大于1的数字；对于单字节字符集，因为所有字符都只占用一个字节，所以这个值将是1。

SHOW CHARACTER SET语句最早出现于MySQL 4.1版本。

#### D.1.44 SHOW COLUMN TYPES

这条语句将列出可以用来创建MySQL数据表的各种数据列类型的信息。它的输出报告由以下输出列组成：

- Type 数据列类型的名称。
- Size 该类型的存储尺寸，以字节为计量单位。
- Min\_value 该类型的取值范围的最小值。
- Max\_value 该类型的取值范围的最大值。
- Prec 该类型的精度。
- Scale 该类型的表示范围因子。



- **Nullable** 该类型是否允许为NULL值。
- **Auto\_Increment** 该类型是否可以用做AUTO\_INCREMENT序列。
- **Unsigned** 该类型是否具备UNSIGNED属性。
- **Zerofill** 该类型是否具备ZEROFILL属性。
- **Searchable** 能否对该类型进行搜索。
- **Case\_Sensitive** 该类型是否要区分字母的大小写情况。
- **Default** 该类型的默认值。根据该类型的数据列是否允许NULL值，有可能出现同一类型有多个默认值的情况。
- **Comment** 一条关于该数据列类型的描述性注释信息。

SHOW COLUMN TYPES语句最早出现于MySQL 4.1版本。

#### D.1.45 SHOW [ FULL ] COLUMNS

这条语句将列出关于给定数据表里各数据列的描述信息。SHOW FIELDS是SHOW COLUMNS的同义词。

```
SHOW COLUMNS FROM president;
SHOW FIELDS FROM president;
SHOW COLUMNS FROM president FROM sampdb;
SHOW FULL COLUMNS FROM tables_priv FROM mysql LIKE '%priv';
```

SHOW COLUMNS语句的输出报告能够提供关于数据表里的每一个数据列的以下信息：

- **Field** 该数据列的名字。
- **Type** 该数据列的类型。在类型名的后面可能还会列出一些相关的属性。
- **Null** Yes表示该数据列允许包含NULL值；空白则表示不允许。
- **Key** 是否有建立在该数据列上的索引。
- **Default** 该数据列的默认值。
- **Extra** 与该数据列有关的其他信息。
- **Privileges** 你在该数据列上的操作权限。只有MySQL 3.23.0及以后的版本能够提供这部分信息。在从MySQL 3.23.0到3.23.32的版本里，这部分信息总是显示出来；但从MySQL 3.23.32版本开始，这部分信息只有在你给出了FULL关键字时才会显示出来。
- **Comment** 在定义该数据列时在COMMENT子句里给出的注释信息。这部分信息只有在MySQL 4.1及以后的版本里才会被显示出来，且只有在给出了FULL关键字时才会被显示出来。

#### D.1.46 SHOW CREATE DATABASE

这条语句将把创建给定数据库所需要的CREATE DATABASE语句显示出来。

```
SHOW CREATE DATABASE sampdb;
```

SHOW CREATE DATABASE语句最早出现于MySQL 4.1版本。

#### D.1.47 SHOW CREATE TABLE

这条语句将把与给定数据表的结构相对应的CREATE TABLE语句显示出来。

```
SHOW CREATE TABLE absence;
```

在默认的情况下，SHOW CREATE TABLE语句所产生的语句里的数据表、数据列和索引的名字都加有引号；这种行为由SQL\_QUOTE\_SHOW\_CREATE选项控制（见本附录前面内容里的SET语句条目）。

SHOW CREATE TABLE语句最早出现于MySQL 3.23.20版本。

#### D.1.48 SHOW DATABASES

这条语句将列出一份MySQL服务器主机上当前可用的数据库名单。

```
SHOW DATABASES;  
SHOW DATABASES LIKE 'test%';
```

如果你不具备SHOW DATABASE权限，就只能看到你有权访问的那些数据库。

#### D.1.49 SHOW GRANTS

这条语句将把给定用户的访问权限列举出来，用户名必须以'user\_name'@'host\_name'的格式给出，这里的user\_name和host\_name与它们在GRANT语句中的语法相同。

```
SHOW GRANTS FOR 'root'@'localhost';  
SHOW GRANTS FOR ''@'cobra.snake.net';
```

SHOW GRANTS语句的输出恰好是相应的GRANT语句所需要的东西——可以用它重新创建出给定账户的权限来。

这条语句最早出现于MySQL 3.23.4版本。

#### D.1.50 SHOW INDEX

这条语句将显示关于给定数据表中的索引的信息。SHOW KEYS是SHOW INDEX的同义词。

```
SHOW INDEX FROM score;  
SHOW KEYS FROM score;  
SHOW INDEX FROM sampdb.score;  
SHOW INDEX FROM score FROM sampdb;
```

这条语句的输出报告由以下输出列组成：

- Table 包含该索引的数据表的名字。
- Non\_unique 1表示该索引允许包含重复的值；0表示不允许。

- **Key\_name** 这个索引的名字。
- **Seq\_in\_index** 数据列在索引中的序号；索引数据列是从1开始编号的。
- **Column\_name** 数据列的名字。
- **Collation** 数据列在索引中的排序顺序；它的可取值是A（升序）、D（降序）或NULL（不排序）。MySQL目前还不支持按降序排序的键字，但未来的版本会增加这一机制。
- **Cardinality** 索引中惟一化键值的个数。以--analyze选项启动执行的myisamchk或isamchk客户程序将刷新MyISAM或ISAM数据表的这个值。ANALYZE TABLE语句将刷新MyISAM或BDB数据表的这个值。
- **Sub\_part** 如果只对数据列的前 $n$ 个字节进行了索引的话，这个输出列将给出该前缀以字节计数的长度。如果是对整个数据列进行了索引的话，这个输出列里的值将是NULL。
- **Packed** 键字的压缩方式，NULL表示没有压缩。
- **Null** YES表示该数据列允许包含NULL值；空白则表示不允许。
- **Index\_type** 数据列的索引方式，比如BTREE或FULLTEXT等。
- **Comment** 保留供该索引的内部注释之用。

Packed和Comment输出列是MySQL 3.23.0版本新增的，Null和Index\_type输出列是4.0.2版本新增的。

#### D.1.51 SHOW INNODB STATUS

这条语句将显示InnoDB数据表处理程序的内部操作状态信息。它最早出现于MySQL 3.23.52版本。

#### D.1.52 SHOW LOGS

这条语句将显示关于MySQL日志文件的信息。但它目前还只能用在BDB日志上。

SHOW LOGS语句的输出报告由以下输出列组成：

- **File** 日志文件的名字。
- **Type** 日志类型。
- **Status** 日志状态；比如IN USE。

这条语句最早出现于MySQL 3.23.29版本。

#### D.1.53 SHOW MASTER LOGS

这条语句要在镜像机制中的主服务器上使用。它将把主服务器上当前可用的二进制日志的名字显示出来。在用SHOW SLAVE STATUS语句查明镜像机制中的各个从服务器都是与哪些二进制日志进行同步的以及它们当前的同步位置之后，在发出PURGE MASTER LOGS语句之前，通常还需要先用SHOW MASTER LOGS语句去查一下主服务器上的日志文件使用情况。

SHOW MASTER LOGS语句最早出现于MySQL 3.23.28版本。

### D.1.54 SHOW MASTER STATUS

这条语句要在镜像机制中的主服务器上使用。它能让你了解主服务器上二进制变更日志的状态信息。

SHOW MASTER STATUS语句的输出报告由以下输出列组成：

- File 二进制变更日志文件名。
- Position MySQL服务器在该文件里的当前写位置。
- Binlog\_do\_db 一份以逗号分隔的、通过--binlog\_do\_db选项明确地与二进制日志建立了镜像关系的数据库名单；如果为空，则表示没有这样的数据库。
- Binlog\_ignore\_db 一份以逗号分隔的、通过--binlog\_ignore\_db选项明确地与二进制日志解除了（或者根本就没有建立）镜像关系的数据库名单；如果为空，则表示没有这样的数据库。

SHOW MASTER STATUS语句最早出现于MySQL 3.23.22版本。

### D.1.55 SHOW PRIVILEGES

这条语句将显示可以由你来进行授权的权限以及它们的含义。

这条语句的输出报告由以下输出列组成：

- Privilege 权限的名字。
- Context 权限的有效范围，比如Server Admin（MySQL服务器的系统管理员）、Databases或Tables等。
- Comment 关于权限用途的描述信息。

这条语句最早出现于MySQL 4.1版本。

### D.1.56 SHOW [ FULL ] PROCESSLIST

这条语句将显示关于那些正在MySQL服务器里运行的线程的信息。它的输出报告由以下输出列组成：

- Id 客户（程序）的线程ID。
- User 与该线程相关联的客户的账户名。
- Host 该客户是从哪一台主机来建立的连接。
- db 该线程的默认数据库。
- Command 该线程正在执行的语句。
- Time 该线程正在执行的语句所花费的时间，以秒为计量单位。
- State MySQL对一条SQL语句的处理过程可以被划分为几个更细致的阶段；这个输出列能告诉我们MySQL正进行到哪一个阶段。如果你想向MySQL的开发者反映使用MySQL时遇到的问题，或者想通过MySQL邮件列表向别人请教为什么你的线程总停留在某个阶段，就需要用这个输出列里的信息。
- Info 正被执行的查询。从MySQL 3.23.7版本开始，在SHOW PROCESSLIST命令里使用

FULL选项将使你查看到Info字段中数据库查询命令的完整文本——如果没有这个选项，就只能看到前100个字符。

#### D.1.57 SHOW SLAVE HOSTS

这条语句要在镜像机制中的主服务器上使用。它能让你了解到当前注册在这个主服务器上（即与之建立了镜像关系）的从服务器的信息。

SHOW SLAVE HOSTS语句的输出报告由以下输出列组成：

- Server\_id 从服务器的ID号。
- Host 从服务器所在的主机名。
- User 从服务器用来建立当前连接的账户。
- Password 从服务器用来建立当前连接的口令。
- Port 从服务器连接在哪个端口上。
- Rpl\_recovery\_rank 镜像恢复级别。
- Master\_id 主服务器的ID号。

User和Password输出列只有在主服务器是以--show-slave-auth-info选项启动时才会出现。

SHOW SLAVE HOSTS语句最早出现于MySQL 4.0版本。

#### D.1.58 SHOW SLAVE STATUS

这条语句要在镜像机制中的从服务器上使用，它将显示主服务器的镜像状态信息。

这条语句的输出报告由以下输出列组成：

- Master\_Host 主服务器的主机名或IP地址。
- Master\_User 用来连接主服务器的用户名。
- Master\_Port 用来连接主服务器的端口号。
- Connect\_retry 在放弃前需要进行的主服务器连接尝试次数。
- Master\_Log\_File 主服务器当前使用的二进制变更日志的文件名。
- Read\_Master\_Log\_Pos 从服务器的I/O线程在主服务器的二进制变更日志里的当前读位置。
- Relay\_Log\_File 从服务器上的中继日志文件的名称。
- Relay\_Log\_Pos 从服务器在当前中继日志中的读位置。
- Relay\_Master\_Log\_File 主服务器上的中继日志文件的名称。
- Slave\_IO\_Running 从服务器的I/O线程是否正在运行。
- Slave\_SQL\_Running 从服务器的SQL线程是否正在运行。
- Replicate\_do\_db 一份以逗号分隔的、通过--replicate-do-db选项明确地表明需要进行镜像处理的数据库名单；如果为空，则表示没有这样的数据库。
- Replicate\_ignore\_db 一份以逗号分隔的、通过--replicate\_ignore\_db选项明确地表明不需要进行镜像处理的数据库名单；如果为空，则表示没有这样的数据库。



- Last\_errno 最近一次的出错代码, 0表示没有出错。
- Last\_error 最近一次的出错信息, 空白表示没有出错。
- Skip\_counter 从服务器需要跳过(即不需要同步)的、来自主服务器的日志事件的个数。
- Exec\_master\_log\_pos 从服务器的SQL线程在主服务器的二进制变更日志里的当前执行位置。
- Relay\_log\_space 中继日志文件的总长度。

SHOW SLAVE STATUS语句最早出现于MySQL 3.23.22版本。

#### D.1.59 SHOW STATUS

这条语句将显示MySQL服务器的状态变量和它们的当前取值。从MySQL 3.23.0版本开始, 如果给SHOW STATUS语句加上一个LIKE 'pattern'子句, MySQL就将只把名字与给定模式相匹配的那些变量显示出来。

下面列出的是一些比较通用的变量。与语句计数器、查询缓存区、SSL连接等问题有关的变量将分别集中在稍后的几个小节里列出。

- Aborted\_clients 因客户(程序)没有正确地关闭之而被丢弃的客户连接的个数。
- Aborted\_connects 试图连接MySQL服务器但没有成功的次数。
- Bytes\_received MySQL服务器从所有客户(程序)那里接收到的字节总数。这个变量最早出现于MySQL 3.23.7版本。
- Bytes\_sent MySQL服务器向所有客户(程序)发送出去的字节总数。这个变量最早出现于MySQL 3.23.7版本。
- Connections 试图连接MySQL服务器的尝试次数(成功或失败的连接尝试都包括在内)。如果这个数字很大, 就应该考虑在客户(程序)里使用永久性连接。
- Created\_tmp\_disk\_tables MySQL服务器在对SQL查询语句进行处理时在磁盘上创建的临时数据表的个数。这个变量最早出现于MySQL 3.23.24版本。
- Created\_tmp\_files MySQL服务器所创建的临时文件的个数。这个变量最早出现于MySQL 3.23.28版本。
- Created\_tmp\_tables MySQL服务器在对SQL查询语句进行处理时在内存里创建的临时数据表的个数。
- Delayed\_errors 在处理INSERT DELAYED数据行时发生的出错的个数。
- Delayed\_insert\_threads INSERT DELAYED处理程序的当前个数。
- Delayed\_writes 已被写入数据库的INSERT DELAYED数据行的个数。
- Flush\_commands 已执行完成的FLUSH语句的个数。
- Handler\_commit 提交一个事务(即让该事务所做的修改生效)的请求的个数。这个变量最早出现于MySQL 4.0.2版本。
- Handler\_delete 从数据表里删除一个数据行的请求的个数。
- Handler\_read\_first 读取索引中的第一个索引项的请求的个数。

- **Handler\_read\_key** 根据某索引值而读取一个数据行的请求的个数。
- **Handler\_read\_next** 按索引顺序读取下一个数据行的请求的个数。
- **Handler\_read\_prev** 按索引逆序读取前一个数据行的请求的个数。这个变量最早出现于MySQL 3.23.6版本。
- **Handler\_read\_rnd** 根据某个数据行的位置而读取该数据行的请求的个数。
- **Handler\_read\_rnd\_next** 读取下一个数据行的请求的个数。如果这个数字很大,就说明有很多查询需要通过全表扫描才能完成或者有很多查询没有使用适当的索引。这个变量最早出现于MySQL 3.23.6版本。
- **Handler\_rollback** 回滚一个事务(即撤销该事务)的请求的个数。这个变量最早出现于MySQL 4.0.2版本。
- **Handler\_update** 对数据表里的一个数据行进行修改的请求的个数。
- **Handler\_write** 往数据表里插入一个数据行的请求的个数。
- **Key\_blocks\_used** 索引缓存区里已被使用的区块的个数。
- **Key\_read\_requests** 从索引缓存区读取一个区块的请求的个数。
- **Key\_reads** 从磁盘读出索引块的物理读操作的次数。
- **Key\_write\_requests** 向索引缓存区写一个区块的请求的个数。
- **Key\_writes** 向磁盘写索引块的物理写操作的次数。
- **Max\_used\_connections** 同时处于打开状态的连接的最大个数。
- **Not\_flushed\_delayed\_rows** INSERT DELAYED查询中尚未被写到磁盘上去的数据行的个数。
- **Not\_flushed\_key\_blocks** 键字缓存区中已经被修改但还没有被写到磁盘上去的区块的个数。
- **Opened\_tables** 曾被打开过的数据表的总数。如果这个数字很大,就应该考虑加大数据表缓存区的尺寸。
- **Open\_files** 当前处于打开状态的文件的个数。
- **Open\_streams** 已被打开的流的个数。“流”是以fopen()函数打开的文件;这个变量只适用于日志文件。
- **Open\_tables** 当前处于打开状态的数据表的个数。
- **Questions** MySQL服务器截止到目前所接收到的查询的个数(成功和不成功的查询都包括在内)。Questions和Update的比值就是MySQL服务器每秒接收到的查询个数。
- **Rpl\_status** 镜像机制的自动防止故障状态。这个变量最早出现于MySQL 4.0.0版本,但还没有使用。
- **Select\_full\_join** 没有使用索引而完成的多数据表关联检索操作的次数。这个变量最早出现于MySQL 3.23.25版本。
- **Select\_full\_range\_join** 利用参照表上的区间搜索操作而完成的多数据表关联检索操作的次数。这个变量最早出现于MySQL 3.23.25版本。
- **Select\_range** 利用第一个数据表上的某个区间而完成的多数据表关联检索操作的次数。这

个变量最早出现于MySQL 3.23.25版本。

- **Select\_range\_check** 必须使用区间搜索操作才能从后续的数据表里检索出有关数据行的多数据表关联检索操作的次数。这个变量最早出现于MySQL 3.23.25版本。
- **Select\_scan** 通过对第一个数据表进行全表扫描而完成的多数据表关联检索操作的次数。这个变量最早出现于MySQL 3.23.25版本。
- **Slave\_open\_tmp\_tables** 镜像机制中的从服务器线程所打开的临时数据表的个数。这个变量最早出现于MySQL 3.23.29版本。
- **Slave\_running** 这个服务器是否正运行作为一个当前正连接到某个服务器的从服务器。这个变量最早出现于MySQL 3.23.16版本。
- **Slow\_launch\_threads** 花费了长于slow\_launch\_time秒的时间才创建出来的线程的个数。这个变量最早出现于MySQL 3.23.15版本。
- **Slow\_queries** 花费了长于long\_query\_time秒的时间才执行完毕的查询的个数。
- **Sort\_merge\_passes** 排序算法为完成排序工作而进行的遍历次数。这个变量最早出现于MySQL 3.23.28版本。
- **Sort\_range** 利用区间而进行的排序操作的次数。这个变量最早出现于MySQL 3.23.25版本。
- **Sort\_rows** 对多少个数据行进行了排序。这个变量最早出现于MySQL 3.23.25版本。
- **Sort\_scan** 利用全表扫描操作而完成的排序操作的次数。这个变量最早出现于MySQL 3.23.25版本。
- **Table\_locks\_immediate** 能够无需等待地立刻得到满足的数据表锁定请求的个数。这个变量最早出现于MySQL 3.23.33版本。
- **Table\_locks\_waited** 必须等待一段时间才能得到满足的数据表锁定请求的个数。如果这个数字很大，就表明系统存在着性能方面的问题。这个变量最早出现于MySQL 3.23.33版本。
- **Threads\_cached** 线程缓存区里现在有多少个线程。这个变量最早出现于MySQL 3.23.17版本。
- **Threads\_connected** 现正处于打开状态的连接的个数。
- **Threads\_created** 截止到目前，为处理客户连接而创建出来的线程的总数。这个变量最早出现于MySQL 3.23.31版本。
- **Threads\_running** 现正处于非休眠状态的线程的个数。
- **Uptime** MySQL服务器自开始运行以来已经持续运行了多长的时间，以秒为计量单位。

#### 1. 与语句计数器有关的状态变量

MySQL 3.23.47及以后的版本增加了一组由服务器负责管理的状态变量，这些状态变量的用途是充当语句计数器，即服务器分别执行了多少条不同类型的语句（命令）。有好几十个语句计数器状态变量，它们都有类似的名称，这里就不把它们逐一地列举出来了。这些语句计数器状态变量的名字全都以“Com\_”开头，然后是一个表明某种具体语句类型的后缀。比如说，Com\_select和Com\_drop\_table变量分别表明MySQL服务器曾经执行过多少条SELECT和DROP TABLE语句。

## 2. 与查询缓存区有关的状态变量

下面这些状态变量都与查询缓存区的操作有关。它们都是从MySQL 4.0.1版本才开始出现的；查询缓存区本身也是从这个版本开始才被新增到MySQL软件里来的。

- `Qcache_free_blocks` 查询缓存区里的可用内存块的个数。
- `Qcache_free_memory` 查询缓存区里的可用内存量。
- `Qcache_hits` 查询缓存区的命中率，即有多少个查询请求是使用存放在这个缓存区里的查询命令来满足的。
- `Qcache_inserts` 截止到目前，在查询缓存区里注册过的查询命令的总数。
- `Qcache_not_cached` 无法被缓存或者因用户使用了`SQL_NO_CACHE`关键字而没有被缓存的查询命令的个数。
- `Qcache_queries_in_cache` 查询缓存区里现在注册有多少条查询命令。
- `Qcache_total_blocks` 查询缓存区里总共有多少个内存块。

注意，`SHOW VARIABLES`语句也能列出一些与查询缓存区有关的变量；这些变量的名字全都以“`query_cache`”开头。

## 3. 与SSL有关的状态变量

下列变量将向你提供各种与SSL管理代码有关的信息。但它们当中有很多只能反映与当前连接有关的情况，如果你使用的不是SSL连接，它们就将是空白的。这些变量最早出现于MySQL 4.0版本，SSL支持机制也是从这个版本开始才增加到MySQL软件里来的。不过，如果你的MySQL服务器在编译时没有包括SSL支持机制，它们将不可用。

- `Ssl_accept_renegotiates` 在服务器模式里开始二次协商（renegotiation）过程的次数。
- `Ssl_accepts` 在服务器模式里开始SSL/TLS握手过程的次数。
- `Ssl_callback_cache_hits` 在服务器模式里从外部会话缓存区成功地检索到的会话的个数。
- `Ssl_cipher` 当前连接所使用的SSL加密协议（若当前不存在生效的加密协议，则为空）。可以利用这个变量来判断当前连接是否是加密的。
- `Ssl_cipher_list` 当前有哪些SSL加密协议可供选用。
- `Ssl_client_connects` 在客户（程序）模式里开始SSL/TLS握手过程的次数。
- `Ssl_connect_renegotiates` 在客户（程序）模式里开始二次协商过程的次数。
- `Ssl_ctx_verify_depth` SSL上下文的验证深度。
- `Ssl_ctx_verify_mode` SSL上下文的验证模式。
- `Ssl_default_timeout` SSL会话默认的超时时间。如果在这段时间内没有客户操作，MySQL将关闭连接。
- `Ssl_finished_accepts` 在服务器模式里成功地建立起来的SSL/TLS会话的个数。
- `Ssl_finished_connects` 在客户（程序）模式里成功地建立起来的SSL/TLS会话的个数。
- `Ssl_session_cache_hits` 在SSL会话缓存区里成功地检索到的SSL会话的个数。
- `Ssl_session_cache_misses` 没能在SSL会话缓存区里成功地检索到的SSL会话的个数。
- `Ssl_session_cache_mode` MySQL服务器所使用的SSL机制的类型。

- `Ssl_session_cache_overflows` 因SSL会话缓存区满而从缓存区去除的会话的个数。
- `Ssl_session_cache_size` SSL会话缓存区的容量，即它最多能够容纳多少个会话。
- `Ssl_session_cache_timeouts` 因超时而从SSL会话缓存区去除的会话的个数。
- `Ssl_session_reused` 当前会话是不是此前某个会话的再次使用。
- `Ssl_used_session_cache_entries` SSL会话缓存区里现在容纳着多少个SSL会话。
- `Ssl_verify_depth` SSL验证深度。
- `Ssl_verify_mode` SSL验证模式。
- `Ssl_version` 当前连接所使用的SSL协议版本。

#### D.1.60 SHOW TABLE STATUS

SHOW TABLE STATUS语句显示关于数据库里的给定数据表的描述信息。

```
SHOW TABLE STATUS;
SHOW TABLE STATUS FROM sampdb;
SHOW TABLE STATUS FROM mysql LIKE '%priv';
```

这条语句的输出报告由以下输出列组成：

- `Name` 数据表的名字。
- `Type` 数据表的类型，如NISAM（ISAM数据表）、MyISAM、HEAP或InnoDB等。
- `Row_format` 数据行的存储格式，如Fixed（固定长度的数据行）、Dynamic（可变长度的数据行）或Compressed（压缩存储的数据行）等。
- `Rows` 数据表里有多少个数据行。对于某些数据表类型——比如BDB和InnoDB，这个数字只是一个大概的估算值。
- `Avg_row_length` 数据表各数据行的平均长度，以字节为计量单位。
- `Data_length` 数据表的数据文件的长度，以字节为计量单位。
- `Max_data_length` 数据表的数据文件所能增长到的最大长度，以字节为计量单位。
- `Index_length` 数据表的索引文件的实际长度，以字节为计量单位。
- `Data_free` 数据表的数据文件中尚未使用的字节数。如果这个数字很大，就应该用OPTIMIZE TABLE语句来优化一下数据表。
- `Auto_increment` 将在数据表的AUTO\_INCREMENT数据列里生成的下一个序列编号值。
- `Create_time` 创建数据表的时间。
- `Update_time` 最近一次对数据表做出修改的时间。
- `Check_time` 最近一次使用myisamchk客户程序对数据表进行检查或修复的时间。如果这个输出列里的值是NULL，则表明从没对数据表进行过检查或修复。
- `Charset` 数据表的字符集。
- `Create_options` 在用来创建数据表的CREATE TABLE语句里的*table\_options*子句里给出的其他选项。
- `Comment` 在创建数据表时给出的任何注释文本。对于InnoDB数据表，这个输出列将给出



数据表上的外键定义；InnoDB数据表空间的可用空间量也将显示在这个输出列里。

SHOW TABLE STATUS语句最早出现于MySQL 3.23.0版本；其输出报告里的Charset输出列是从MySQL 4.1版本开始新增加的。

#### D.1.61 SHOW TABLE TYPES

SHOW TABLE TYPES语句将列举出一份当前MySQL服务器所支持的数据表处理程序的清单。这条语句的输出报告由以下输出列组成：

- Type 数据表类型（MyISAM、InnoDB等等）。
- Support MySQL服务器对该数据表类型的支持程度：YES表示支持、NO表示不支持、DISABLED表示虽然支持但相关功能在服务器启动时被禁用、DEFAULT表示这种类型是MySQL服务器的默认数据表类型。
- Comment 关于数据表类型的描述性文本。

SHOW TABLE TYPES语句最早出现于MySQL 4.1版本。

#### D.1.62 SHOW [ OPEN ] TABLES

SHOW TABLES语句将列举出一份给定数据库里的非TEMPORARY数据表的清单。

```
SHOW TABLES;
SHOW TABLES FROM sampdb;
SHOW TABLES FROM mysql LIKE '%priv';
```

SHOW OPEN TABLES语句（从MySQL 3.23.33版本开始）将把当前已注册在数据表缓存区里（即已经被打开）的数据表列举出来；SHOW OPEN TABLES语句中的FROM和LIKE子句将被忽略。

```
SHOW OPEN TABLES;
```

#### D.1.63 SHOW [ GLOBAL | SESSION ] VARIABLES

这条语句将把MySQL服务器变量和它们的值显示出来。

```
SHOW VARIABLES;
SHOW VARIABLES LIKE '%thread%';
```

从MySQL 4.0.3版本开始，这条语句既可以用来查看给定变量的全局级（在整个服务器的范围内生效）取值，也可以用来查看它的会话级（只在某次特定的会话过程中生效）取值。在默认的情况下，这条语句将显示给定变量的会话级取值（只有在其会话级取值不存在的时候才显示变量的全局级取值）；如果想查看给定变量的全局级取值，就需要明确地给出一个限定字，如下所示（变量的会话级取值也可以用这个办法来查看）：

```
SHOW GLOBAL VARIABLES [LIKE pattern'];
SHOW SESSION VARIABLES [LIKE pattern'];
```

LOCAL是SESSION的同义词。动态变量的取值还可以利用SELECT语句来查看，如下所示：

```
SELECT @@GLOBAL.table_type, @@SESSION.table_type, @@LOCAL.table_type;
```

利用SELECT语句来查看动态变量取值的好处是可以把查询结果（即变量的取值）方便地用在各种有关的上下文里。

SHOW VARIABLES语句能让我们查看到下列变量的取值情况。除少数后来新增的变量（我们将在有关条目里特别注明）以外，大多数变量从MySQL 3.22.0版本开始就已经存在于MySQL软件里了。（大多数InnoDB变量最早出现于MySQL 3.23.29版本，它们当时的名字全都以innobase\_开头——从MySQL 3.23.27版本开始，它们的名字才改为以innodb\_开头。）此外，有些变量只有在特定的配置情况下才会出现在SHOW VARIABLES语句的输出报告里。比如说，以bdb\_开头的变量只有在MySQL服务器里有BDB数据表处理程序组件时才会出现在SHOW VARIABLES语句的输出报告里。

从MySQL 4.0.3版本开始，有些服务器变量允许你在MySQL服务器仍在运行时动态地加以改变（具体做法见前面内容里的SET语句条目）。在以下内容里，对于这些允许进行动态修改的变量，我们将在其变量名的后面用“（全局）”和/或“（会话）”来注明MySQL都允许在哪一个级别对该变量进行动态修改。（注意：因为MySQL 4.0.3之前的版本不支持服务器变量的动态修改机制，所以如果你的MySQL版本低于4.0.3，变量名后面的“（全局）”和/或“（会话）”标注将没有任何意义。）

- **back\_log** 在MySQL服务器正在处理一个连接请求的时候，如果又有其他客户（程序）发来连接请求，那些后到的连接请求将排入一个队列等待MySQL服务器进行处理。这个变量给出的是这个队列所能容纳的待处理连接请求的最大个数。
- **ansi\_mode** MySQL服务器在启动时是否使用了--ansi选项。这个变量最早出现于MySQL 3.23.6版本，但从MySQL 3.23.41版本开始，因为新增了sql\_mode变量，这个变量又被弃用了。
- **basedir** MySQL软件根安装目录的路径名。
- **bdb\_cache\_size** BDB数据表处理程序用来缓存数据行和索引行的缓存区的长度。如果在启动MySQL时使用了--skip-bdb选项，就将禁用BDB处理程序并把这个变量的值设置为0，从而减少了内存占用量。这个变量最早出现于MySQL 3.23.14版本。
- **bdb\_home** BDB主目录的路径名；它通常与datadir变量有着相同的值。这个变量最早出现于MySQL 3.23.14版本。
- **bdb\_log\_buffer\_size** 供BDB事务日志使用的缓存区的长度。这个变量最早出现于MySQL 3.23.21版本。
- **bdb\_logdir** BDB日志目录的路径名，BDB处理程序将把日志文件写到这个目录里去。这个变量最早出现于MySQL 3.23.14版本。
- **bdb\_max\_lock** 允许同时施加在同一个BDB数据表上的数据锁的最大个数。这个变量最早出现于MySQL 3.23.29版本。
- **bdb\_shared\_data** 表明BDB是否是在多进程模式下启动的。这个变量最早出现于MySQL

3.23.29版本。

- **bdb\_tmpdir** BDB临时目录的路径名，BDB处理程序将在这个目录里创建临时文件。这个变量最早出现于MySQL 3.23.14版本。
- **bdb\_version** BDB处理程序的版本号。这个变量最早出现于MySQL 3.23.31版本。
- **binlog\_cache\_size** (全局) 用来临时存放那些构成某次事务的SQL语句的缓存区的长度。当这个缓存区被填满的时候，缓存在其中的SQL语句将被写到有关的二进制日志里去。(这只发生在事务最终得到了提交的场合。对于那些最终被回滚的事务，构成事务的各个SQL语句将被丢弃。) 这个变量最早出现于MySQL 3.23.29版本。
- **bulk\_insert\_buffer\_size** (全局，会话) 用来对往MyISAM数据表里批量插入多个新数据行的语句——如LOAD DATA语句、一次插入多个数据行的INSERT语句、INSERT INTO ... SELECT语句等等——进行优化的缓存区的长度。如果把这个变量设置为0，MySQL就将不对这些语句进行优化。  
这个变量最早出现于MySQL 4.0.0版本，它当时的名字叫myisam\_bulk\_insert\_tree\_size；但从MySQL 4.0.3版本开始，这个变量被重新命名为bulk\_insert\_buffer\_size。
- **character\_set** 默认字符集的名字。这个变量最早出现于MySQL 3.23.3版本。
- **character\_sets** MySQL服务器支持使用的字符集清单，各字符集名称之间用空格分隔。这个变量最早出现于MySQL 3.23.15版本。
- **concurrent\_insert** (全局) 当MySQL服务器正在处理某个MyISAM数据表上的SELECT查询时，如果又接收到对这个数据表的INSERT查询，MySQL将根据这个变量的值来决定是否要同时进行这个后来的INSERT查询。这个变量的默认设置是激活这种并发处理机制，但可以用--skip-concurrent-insert选项来禁用之。这个变量最早出现于MySQL 3.23.7版本。
- **connect\_timeout** (全局) 连接超时时间；在最初的连接握手阶段，如果mysqld在等待了这么长的时间后还没有收到来自客户（程序）的数据包，就将放弃对这个连接的处理。
- **convert\_character\_set** (会话) 用来对来自和发送给客户（程序）的字符串进行转换的字符集的名字。这个变量最早出现于MySQL 4.0.3版本。
- **datadir** MySQL数据目录的路径名。
- **delayed\_insert\_limit** (全局) 在处理INSERT DELAYED语句的时候，MySQL每插入delayed\_insert\_limit个数据行，就会去看看是否有新到的SELECT语句正排队等待着对有关的数据表进行检索：如果没有，则继续插入数据行；如果有，则让检索操作先执行。
- **delayed\_insert\_timeout** (全局) 在处理INSERT DELAYED语句的时候，当把队列（参见下一条目）里的数据行全都插入到有关的数据表里之后，MySQL会等待delayed\_insert\_timeout秒，看有没有新的INSERT DELAYED数据行到达：如果有，则继续进行插入操作；如果没有，则结束这次插入操作。
- **delayed\_queue\_size** (全局) 在被实际插入到各有关数据表里去之前，INSERT DELAYED数据行将在一个队列里等待MySQL来处理它们；delayed\_query\_size就是这个队列所能容纳的数据行的最大个数。当这个队列满时，后续的INSERT DELAYED语句将

被阻塞直到这个队列里有容纳它们的空间为止。

- **delay\_key\_write (全局)** 对于使用了DELAY\_KEY\_WRITE选项创建出来的MyISAM数据表, MySQL将根据这个变量的设置情况来决定是否延缓键字的写操作。如果这个变量的取值是ON (这是该变量的默认值), MySQL服务器就将根据MyISAM数据表的DELAY\_KEY\_WRITE选项而采取相应的行动 (如果在创建数据表时使用了DELAY\_KEY\_WRITE = 1选项, 则延缓数据表上的键字写操作; 如果在创建数据表时使用了DELAY\_KEY\_WRITE = 0选项, 则不延缓数据表上的键字写操作)。如果这个变量的取值是OFF, 则不延缓任何一个数据表 (不管它在创建时是如何声明的) 上的键字写操作。如果这个变量的取值是ALL, 则延缓任何一个数据表 (不管它在创建时是如何声明的) 上的键字写操作。  
这个变量最早出现于MySQL 3.23.5版本, 它当时的名字叫delayed\_key\_write; 但从MySQL 3.23.8版本开始, 这个变量被重新命名为delay\_key\_write。
- **flush (全局)** MySQL服务器是否要在每个修改操作完成后立刻刷新数据表。这个变量最早出现于MySQL 3.22.9版本, 但在Windows系统上却只能用在从3.22.9到3.23.11版本里。
- **flush\_time (全局)** 如果这个变量是一个非零值, 那么每隔flush\_time秒, MySQL就会对数据表进行一次刷新 (先关闭, 再重新打开它们), 以便把尚未写入磁盘的修改写入磁盘。这个变量在UNIX系统上的默认值是0, 在Windows系统上的默认值是1800 (30分钟); 可以通过--flush选项来改变这个变量的值。这个变量最早出现于MySQL 3.22.18版本。
- **ft\_boolean\_syntax** IN BOOLEAN MODE模式下的FULLTEXT搜索所允许使用的各种操作符。这个变量最早出现于MySQL 4.0.1版本。
- **ft\_max\_word\_len** 允许包括在FULLTEXT索引里的单词的最大长度; 大于这个长度的单词将被忽略。如果改变这个变量的值, 就必须重建由这个服务器所管理的所有数据表里的FULLTEXT索引。这个变量最早出现于MySQL 4.0.0版本。
- **ft\_max\_word\_len\_for\_sort** FULLTEXT索引中适合快速排序的单词的最大长度, 只有长度小于这个数字的单词才会被ALTER TABLE、CREATE INDEX、REPAIR TABLE等语句用来建立FULLTEXT索引的快速索引创建方法认为是足够短的。大于这个长度的单词将使用另一种较慢的方法来插入。这个变量最早出现于MySQL 4.0.0版本。
- **ft\_min\_word\_len** 允许包括在FULLTEXT索引中的单词的最小长度; 短于这个长度的单词将被忽略。如果改变这个变量的值, 就必须重建由这个服务器所管理的所有数据表里的FULLTEXT索引。这个变量最早出现于MySQL 4.0.0版本。
- **have\_bdb** YES表示BDB数据表处理程序存在且已被激活; DISABLED表示BDB数据表处理程序存在但被禁用; NO表示BDB数据表处理程序不存在。这个变量最早出现于MySQL 3.23.30版本。
- **have\_innodb** YES表示InnoDB数据表处理程序存在且已被激活; DISABLED表示InnoDB数据表处理程序存在但被禁用; NO表示InnoDB数据表处理程序不存在。  
这个变量最早出现于MySQL 3.23.30, 它当时的名字叫have\_innobase; 但从3.23.37版本开始, 该变量被重新命名为have\_innodb。



- **have\_iasm** YES表示ISAM数据表处理程序存在且已被激活；DISABLED表示ISAM数据表处理程序存在但被禁用；NO表示ISAM数据表处理程序不存在。这个变量最早出现于MySQL 3.23.30版本。
- **have\_openssl** YES表示MySQL服务器支持使用SSL的加密连接；NO表示不支持。这个变量最早出现于MySQL 3.23.43版本，但在实现有SSL支持机制的MySQL 4.0.0版本出现之前并没有实际意义。
- **have\_query\_cache** YES表示查询缓存区可用；NO表示不可用。这个变量最早出现于MySQL 4.0.2版本。
- **have\_raid** YES表示MySQL服务器支持使用与RAID技术有关的CREATE TABLE选项；NO表示不支持。这个变量最早出现于MySQL 3.23.30版本。
- **have\_symlink** YES表示MySQL服务器的符号链接支持机制已激活；NO表示未激活。这个变量最早出现于MySQL 4.0.0版本。
- **init\_file** 在启动MySQL服务器时以--init-file选项给出的一个文件名，MySQL服务器在启动时将自动执行该文件里的SQL语句。这个变量最早出现于MySQL 3.23.2版本。
- **innodb\_additional\_mem\_pool\_size** InnoDB内存池的大小，其中存放着各种供内部使用的数据结构。这个变量最早出现于MySQL 3.23.37版本。
- **innodb\_buffer\_pool\_size** InnoDB缓存区的大小，其中存放着InnoDB数据表的数据和索引。这个变量最早出现于MySQL 3.23.37版本。
- **innodb\_data\_file\_path** InnoDB表空间组件文件的定义。这个变量最早出现于MySQL 3.23.37版本。
- **innodb\_data\_home\_dir** 用于存放InnoDB表空间组件的目录路径名。这个变量最早出现于MySQL 3.23.37版本。
- **innodb\_fast\_shutdown** InnoDB是否将使用它的快速关机方法，这个方法省略了它在正常关机时会进行的几个操作步骤。这个变量最早出现于MySQL 3.23.44版本。
- **innodb\_file\_io\_threads** InnoDB使用的I/O线程的数量。这个变量最早出现于MySQL 3.23.37版本。
- **innodb\_flush\_log\_at\_trx\_commit** 这个变量控制着InnoDB日志在提交事务时的刷新机制，如下表所示：

取 值	含 义
0	每隔一秒写一次日志，同时刷新相应的磁盘文件
1	每提交一次事务写一次日志，同时刷新相应的磁盘文件
2	每提交一次事务写一次日志，但每隔一秒刷新一次相应的磁盘文件

这个变量最早出现于MySQL 3.23.37版本。

- **innodb\_flush\_method** InnoDB用来刷新日志的方法。这个变量最早出现于MySQL 3.23.39版本。
- **innodb\_force\_recovery** 这个变量的值通常是0，但可以被设置为从1~6的某个值以使MySQL服务器即使在InnoDB恢复失败的情况下也能在崩溃后再次启动。关于这个变量的



用途和用法的详细说明请参见本书第13章中的有关内容。这个变量最早出现于MySQL 3.23.44版本。

- `innodb_lock_wait_timeout` 在准备进行一次事务的时候，如果InnoDB在等待了`innodb_lock_wait_timeout`秒后还没有获得有关的数据锁，InnoDB就将回滚这次事务。这个变量最早出现于MySQL 3.23.37版本。
- `innodb_log_arch_dir` 这个变量目前尚未使用。这个变量最早出现于MySQL 3.23.37版本。
- `innodb_log_archive` 这个变量目前尚未使用。这个变量最早出现于MySQL 3.23.37版本。
- `innodb_log_buffer_size` InnoDB事务日志缓冲区的大小。这个变量最早出现于MySQL 3.23.37版本。
- `innodb_log_files_in_group` InnoDB操作所涉及的日志文件的个数。`innodb_log_files_in_group`和`innodb_log_file_size`的乘积就是InnoDB日志的总长度。这个变量最早出现于MySQL 3.23.37版本。
- `innodb_log_file_size` InnoDB日志文件的长度。`innodb_log_files_in_group`和`innodb_log_file_size`的乘积就是InnoDB日志的总长度。这个变量最早出现于MySQL 3.23.37版本。
- `innodb_log_group_home_dir` InnoDB日志目录的路径名，InnoDB日志文件将被写到这个目录里去。这个变量最早出现于MySQL 3.23.37版本。
- `innodb_mirrored_log_groups` InnoDB日志文件组的个数，这个变量的值应该永远是1。这个变量最早出现于MySQL 3.23.37版本。
- `innodb_thread_concurrency` InnoDB操作所涉及的线程的最大个数。这个变量最早出现于MySQL 3.23.44版本。
- `interactive_timeout` (全局, 会话) 如果某个交互式的客户连接在`interactive_timeout`秒内没有操作动作，MySQL服务器就将认为该客户连接不再有保留的必要并自动关闭这个连接。对于非交互式的客户连接，MySQL服务器将使用`wait_timeout`变量的值作为这种超时等待的秒数。这个变量最早出现于MySQL 3.23.7版本。
- `join_buffer_size` (全局, 会话) 全关联缓冲区（即没有使用索引而进行的关联操作所使用的缓冲区）的大小。  
这个变量在MySQL 3.23之前的版本里叫做`join_buffer`。
- `key_buffer_size` (全局) 索引块所使用的缓冲区的大小。这个缓冲区是由连接处理程序线程所共享的。  
这个变量在MySQL 3.23之前的版本里叫做`key_buffer`。
- `language` 用来显示出错信息的语言。这个变量的值可以是某种语言的名称，也可以是包含着该语言各相关文件的某个目录的路径名。
- `large_files_support` MySQL服务器在建立（编译）时是否带有大尺寸文件支持机制。这个变量最早出现于MySQL 3.23.28版本。
- `local_infile` (全局) 是否允许在LOAD DATA语句里使用LOCAL关键字。这个变量最早出现于MySQL 4.0.3版本。

- **locked\_in\_memory** MySQL服务器在内存里是否会被锁定。这个变量最早出现于MySQL 3.23.25版本。
- **log** 查询日志机制是否已被激活。
- **log\_bin** 二进制变更日志机制是否已被激活。这个变量最早出现于MySQL 3.23.14版本。
- **log\_slave\_updates** 镜像机制中的从服务器是否会把接收自主服务器的数据修改操作记录到日志里。如果在某个从服务器上激活了这种日志机制，就可以让这个从服务器在一个链式镜像配置中充当另一个从服务器的主服务器。这个变量最早出现于MySQL 3.23.17版本。
- **log\_slow\_queries** 慢查询日志机制是否已被激活。这个变量最早出现于MySQL 4.0.2版本。
- **log\_update** 变更日志机制是否已被激活（即数据库数据的修改操作是否会被记录到相关的日志里去）。这个变量最早出现于MySQL 3.22.18版本。
- **log\_warnings**（全局，会话）非致命错误所产生的警告性信息是否会被记录到错误日志里去。这个变量最早出现于MySQL 4.0.3版本。
- **long\_query\_time**（全局，会话）如果某个查询命令的执行时间大于long\_query\_time秒，MySQL服务器就将认为它是一个“慢”查询。每出现一个慢查询，Slow\_queries变量的值就会增加一个1；此外，如果慢查询日志机制已被激活，这个查询就将被记录到相应的日志里去。
- **lower\_case\_table\_names** 数据表的名字在被保存到磁盘上去的时候是否会被强制转换为小写字母（从MySQL 4.0.2版本开始，数据库的名字也会受到这个变量的影响）。这个变量最早出现于MySQL 3.23.6版本。
- **low\_priority\_updates**（全局，会话）MySQL服务器在启动时是否使用了--low-priority-updates选项（如果使用了这个选项，数据修改操作的优先级将低于数据检索操作）。这个变量最早出现于MySQL 3.22.5版本。
- **max\_allowed\_packet**（全局，会话）MySQL服务器与客户（程序）之间进行通信时使用的缓冲区的最大长度。这个缓冲区的初始长度是net\_buffer\_length个字节，但可以根据需要扩大到max\_allowed\_packet个字节。在MySQL 4及以后的版本里，max\_allowed\_packet的最大值是1GB；在MySQL 4之前的版本里，这个最大值是16MB。
- **max\_binlog\_cache\_size**（全局）二进制日志缓存区的最大长度。对于构成同一个事务的各条SQL语句来说，这个值就是它们总组合长度的上限。这个变量最早出现于MySQL 3.23.29版本。
- **max\_binlog\_size**（全局）二进制日志的最大长度。如果某个日志达到了这个长度，该日志就将被轮转。这个变量的取值范围是从1KB~1GB。这个变量最早出现于MySQL 3.23.33版本。
- **max\_connections**（全局）允许同时处于打开状态的客户连接的最大个数。
- **max\_connect\_errors**（全局）失败连接的最大允许值；如果来自某个主机的失败连接已经达到了max\_connect\_errors个，来自该主机的其他连接尝试就将被阻塞。之所以要采取这样的做法，是为了预防出现有人正试图从那台主机攻击数据库系统。FLUSH HOSTS语句或mysqladmin flush-hosts命令都可以用来清除主机缓存区以重新激活被阻塞的主机。

- `max_delayed_threads`（全局） 为处理INSERT DELAYED语句而允许创建的线程的最大个数。如果已经创建了这么多线程可又有新的INSERT DELAYED语句到来，新到的INSERT DELAYED语句将被当做非DELAYED语句来处理。

这个变量最早出现于MySQL 3.22.15版本，它当时的名字叫`max_delayed_insert_threads`；但从3.23.0版本开始，这个变量被重新命名为`max_delayed_threads`。

- `max_heap_table_size`（全局，会话） HEAP数据表的最大允许长度。这个变量能够有效地预防MySQL服务器消耗过多的内存。这个变量最早出现于MySQL 3.23.0版本。
- `max_join_size`（全局，会话） 在对多个数据表进行合并关联检索的时候，MySQL优化器会先对将被筛选的数据行的组合个数做一个估算；如果估算值超过了`max_join_size`个数据行，就不进行这次检索而直接返回一条出错信息。这可以有效地预防因出现用户编写的SELECT查询不合理而毫无意义地返回大量数据行的情况。

- `max_sort_length`（全局，会话） MySQL将使用BLOB或TEXT值的前`max_sort_length`个字节对它们进行排序。

- `max_tmp_tables`（全局，会话） 允许同一个客户（程序）同时打开的临时数据表的最大个数。这个变量最早出现于MySQL 3.23.0版本，但现在已经被弃用了。

- `max_user_connections`（全局） 允许同一个用户账户同时打开的客户连接的最大个数。这个变量的默认值是0，意思是“没有限制”；但各账户所打开的客户连接的总数不能超过`max_connections`变量的值。这个变量最早出现于MySQL 3.23.34版本。

允许某给定账户同时打开的客户连接的上限个数可以用GRANT语句来设定。

- `max_write_lock_count`（全局） 在使用了`max_write_lock_count`个写锁定之后，MySQL服务器将适当提升申请读锁定的查询的优先级，即将优先执行读操作性质的查询。这个变量最早出现于MySQL 3.23.7版本。

- `myisam_bulk_insert_tree_size` 参见`bulk_insert_buffer_size`条目。

- `myisam_max_extra_sort_file_size`（全局，会话） MyISAM数据表处理程序将参考这个变量值来决定需要在何种情况下采用一种速度较慢但更安全的键字缓存区索引创建方法。这个变量最早出现于MySQL 3.23.37版本。在MySQL 4.0.3及以后的版本里，这个变量的值以字节为计量单位；在MySQL 4.0.3之前，以MB为计量单位。

- `myisam_max_sort_file_size`（全局，会话） 在MyISAM数据表上，因执行REPAIR TABLE、ALTER TABLE或LOAD DATA等语句而导致的索引重建工作既可以使用临时文件去完成，也可以使用键字缓存区去完成。MySQL将根据这个变量的值来决定使用哪一种方法：如果临时文件的估算长度大于这个值，MySQL就会使用键字缓存区去重建各有关的索引。

这个变量最早出现于MySQL 3.23.37版本。在MySQL 4.0.3及以后的版本里，这个变量的值以字节为计量单位；在MySQL 4.0.3之前，以MB为计量单位。

- `myisam_recover_options` MySQL服务器在启动时使用的--myisam-recover选项的值；这个选项设定了MyISAM数据表的自动检查模式。这个变量最早出现于MySQL 3.23.36版本。

- **myisam\_sort\_buffer\_size** (全局, 会话) 在ALTER TABLE、CREATE INDEX或REPAIR TABLE操作期间, MySQL为完成MyISAM数据表上的索引重新排序工作而分配的缓冲区的长度。这个变量最早出现于MySQL 3.23.16版本。
- **named\_pipe** 命名管道支持机制是否已被激活; 适用于基于Windows NT的MySQL服务器。这个变量最早出现于MySQL 3.23.50版本。(在MySQL 3.23.50之前的版本里, 支持命名管道的MySQL服务器的命名管道支持机制是默认激活的。)
- **net\_buffer\_length** (全局, 会话) MySQL服务器与客户(程序)之间进行通信时使用的缓冲区的初始长度。根据具体情况, 这个缓冲区的长度可以扩大到max\_allowed\_packet个字节。
- **net\_read\_timeout** (全局, 会话) 在MySQL服务器接收客户数据的场合, 如果MySQL服务器在等待了net\_read\_timeout秒之后仍未收到来自客户连接的数据, 就将产生一个读操作超时错误。这个变量最早出现于MySQL 3.23.20版本。
- **net\_retry\_count** (全局, 会话) 在MySQL服务器接收客户数据的场合, 如果读操作因某种原因而中断, MySQL服务器将重试该操作; net\_retry\_count变量给出的就是这种重试的次数。这个变量最早出现于MySQL 3.23.7版本。
- **net\_write\_timeout** (全局, 会话) 在MySQL服务器往客户发送数据的场合, 如果MySQL服务器在经过了net\_write\_timeout秒之后仍未收到来自客户的响应, 就将产生一个写操作超时错误。这个变量最早出现于MySQL 3.23.20版本。
- **open\_files\_limit** 如果取值为非零, 这个变量就是MySQL服务器试图保留的文件描述符的个数。如果这个变量的值是零, MySQL服务器就将以max\_connections \* 5和max\_connections + table\_cache \* 2中较大的那个值作为其试图保留的文件描述符的个数。这个变量最早出现于MySQL 3.23.30版本。
- **pid\_file** 这个变量给出的是一个文件路径名, MySQL服务器将把自己的进程ID号写到这个文件里去。这个变量最早出现于MySQL 3.22.23版本。
- **port** MySQL服务器在其上监听客户连接的TCP/IP端口号。
- **protocol\_version** MySQL服务器所使用的客户/服务器协议的版本号。这个变量最早出现于MySQL 3.22.18版本。
- **query\_cache\_limit** (全局) 查询结果的最大缓存长度; 超过这一长度的查询结果将不会被缓存。这个变量最早出现于MySQL 4.0.1版本。
- **query\_cache\_size** (全局) 用来临时存放查询结果的缓存区的最大长度。把这个变量设置为0将禁用查询缓存机制。这个变量最早出现于MySQL 4.0.1版本。
- **query\_cache\_type** (全局, 会话) 查询缓存区的操作模式, 如下表所示:

模 式	含 义
OFF	不进行缓存
ON	进行缓存, 但以SELECT SQL_NO_CACHE开头的查询命令不包括在内
DEMAND	只对以SELECT SQL_CACHE开头的查询命令进行缓存



这个变量最早出现于MySQL 4.0.1版本，它当时的名字叫query\_cache\_startup\_type；但从4.0.3版本开始，这个变量被重新命名为query\_cache\_type。

- read\_buffer\_size（全局，会话）对数据表做顺序扫描的线程所使用的缓冲区的长度。如有必要，每个客户（程序）都能分配到一个这样的缓冲区。

在MySQL 4.0.3之前的版本里，这个变量的名字是record\_buffer。

- read\_rnd\_buffer\_size（全局，会话）在对数据行进行排序之后，用来顺序读取各数据行的缓冲区的长度。如有必要，每个客户（程序）都能分配到一个这样的缓冲区。

这个变量最早出现于MySQL 3.23.41版本，它当时的名字叫record\_rnd\_buffer；但从4.0.3版本开始，这个变量被重新命名为read\_rnd\_buffer\_size。

- record\_buffer 参见read\_buffer\_size条目。
- record\_rnd\_buffer 参见read\_rnd\_buffer\_size条目。
- rpl\_recovery\_rank（全局）MySQL服务器的镜像恢复级别。这个变量目前尚未投入使用；它的预定用途是允许镜像进制中的某个从服务器在与原先的主服务器失去联系之后能够从那些仍与之有联系的其他镜像服务器当中选择一个作为新的主服务器。这个变量最早出现于MySQL 4.0.0版本。
- safe\_show\_database 是否无条件地显示数据库的名字。如果这个变量的取值是ON，用户就将只能查看到他有权查看的数据库或数据表的名字。如果这个变量的取值是OFF，任何用户将都能查看到所有的数据库的名字。这个变量最早出现于MySQL 3.23.30版本。从MySQL 4.0.3版本开始，如果你想让某位用户查看到所有的数据库的名字，还必须明确地把SHOW DATABASES权限授予给他才行。MySQL 4.0.5及以后的版本没有提供这个变量。
- server\_id（全局）MySQL服务器的镜像ID编号。这个变量最早出现于MySQL 3.23.26版本。
- skip\_external\_locking 外部锁定机制（即文件系统级的锁定机制）是否被抑制。  
在MySQL 4.0.3之前的版本里，这个变量的名字是skip\_locking。
- skip\_locking 参见skip\_external\_locking条目。
- skip\_networking OFF表示允许TCP/IP连接；ON表示禁用TCP/IP连接。在后一种场合，客户（程序）将只能从本地主机通过套接字（如果你使用的是UNIX系统的话）或命名管道（如果你使用的是Windows系统的话）进行连接。这个变量最早出现于MySQL 3.22.23版本。
- skip\_show\_database 如果这个变量的取值是ON，就只有那些拥有SHOW DATABASES权限的用户才能查看数据库的名字；如果这个变量的取值是OFF，则没有这种限制。这个变量最早出现于MySQL 3.23.4版本。
- slave\_net\_timeout（全局）如果镜像机制中的从服务器在经过slave\_net\_timeout秒之后仍未接收到来自主服务器的数据，就将产生一个超时错误。这个变量最早出现于MySQL 3.23.40版本。
- slow\_launch\_time（全局）如果某个线程用了多于slow\_launch\_time秒的时间才被创建出来，它就会被认为是一个“慢创建”线程并将导致状态计数器Slow\_launch\_threads加上一



个1。这个变量最早出现于MySQL 3.23.15版本。

- `socket` UNIX域套接字的路径名或Windows系统中的命名管道的名字。
- `sort_buffer_size` (全局, 会话) 供那些用来完成排序操作的线程 (GROUP BY或ORDER BY) 使用的缓冲区的长度。如有必要, 每个客户 (程序) 都能分配到一个这样的缓冲区。一般来说, 如果你有许多个客户 (程序) 会在同一时间进行排序操作, 就不应该把这个值设置得很大 (超过1MB)。

在MySQL 4.0.3之前的版本里, 这个变量的名字是`sort_buffer`。

- `sql_mode` `--sql-mode`选项的设置值。这个变量最早出现于MySQL 3.23.41版本; 它取代了`ansi_mode`变量。
- `table_cache` (全局) 能够被同时打开的数据表的最大个数。这个缓存区是由全体线程共享的。
- `table_type` (全局, 会话) 默认的数据表类型。如果在创建数据表的时候没有给出`TYPE = type_name`选项或者给出的是一个不受MySQL服务器支持的`type_name`值, 数据表就将被创建为本变量所设定的类型。这个最早出现于MySQL 3.23.0版本。
- `thread_cache_size` (全局) 线程缓存区所能容纳的线程的最大个数。这个变量最早出现于MySQL 3.23.16版本。
- `thread_concurrency` 这个变量只适用于Solaris系统。这个值将被传递给`thr_concurrency()`函数, Solaris系统上的线程管理器将参照这个值来决定应该同时运行多少个MySQL线程。这个变量最早出现于MySQL 3.23.7版本。
- `thread_stack` 各线程的堆栈的长度。
- `timezone` MySQL服务器的地理时区设置。这个变量最早出现于MySQL 3.23.15版本。
- `tmpdir` MySQL服务器将在这个目录里创建临时文件。这个变量最早出现于MySQL 3.22.4版本。在从3.22.0到3.22.3版本里, 它的名字是`tmp_dir`。
- `tmp_table_size` (全局, 会话) 临时数据表以字节计算的最大允许长度。在MySQL 3.23版本之前, 如果临时数据表的长度超过了这个值, MySQL服务器就会把它转换为一个MyISAM数据表并保存到磁盘上去。如果你有足够多的内存的话, 加大这个变量的值将使MySQL服务器能够创建和管理尺寸更大的临时数据表。
- `transaction_isolation` 参见`tx_isolation`条目。
- `tx_isolation` (全局, 会话) 默认的事务隔离级别。  
这个变量最早出现于MySQL 3.23.36版本, 它当时的名字叫`transaction_isolation`; 但从4.0.3版本开始, 这个变量被重新命名为`tx_isolation`。
- `version` 服务器的版本。这个变量的值由一个版本编号以及一个或者多个后缀构成, 各种可用的后缀及其含义可以在本书的附录C对`VERSION()`函数的介绍中查到。
- `wait_timeout` (全局, 会话) 如果某个非交互式的客户连接在`wait_timeout`秒内没有操作动作, MySQL服务器就认为该客户连接不再有保留的必要并自动关闭这个连接。对于交互式的客户连接, MySQL服务器将使用`interactive_timeout`变量的值作为这种超时等待的秒数。

## D.1.64 SLAVE

语法: `SLAVE { START | STOP } [ slave_options ]`

这条语句控制着镜像机制中的从服务器的启停操作。SLAVE START启动从服务器线程, SLAVE STOP则终止之。

这条语句最早出现于MySQL 3.23.16版本。从MySQL4.0.2版本开始, 这条语句增加了一个可选的*slave\_options*子句。该子句由以下选项中的一个或者多个构成, 它们彼此之间要用逗号分隔开:

- `IO_THREAD` 启动或者终止I/O线程, I/O线程负责从主服务器获得SQL查询命令并把它存放中继日志里去。
- `SQL_THREAD` 启动或者终止SQL线程, SQL线程负责从中继日志里读出SQL查询命令并执行它们。

## D.1.65 TRUNCATE

语法: `TRUNCATE [ TABLE ] tbl_name`

这条语句能够快速清空数据表的内容; 其具体做法是先丢弃、再重新创建数据表。这比逐个地删除各个数据行的做法要快捷。

这条语句不支持事务处理机制; 当你在某次事务的过程中或者在已明确锁定的数据表上发出一条TRUNCATE TABLE语句时, MySQL将报告出错。

这条语句最早出现于MySQL 3.23.28版本; 但在3.23.33之前的版本里, TABLE关键字必须省略。

## D.1.66 UNION

语法: `select_statement`

`UNION [ALL] select_statement`

`[UNION select_statement] ...`

UNION其实并不是一条独立的语句, 它只是SELECT语句的一种组合方式, 即把各SELECT语句的查询结果依次追加合并在一起。每个SELECT语句的结果集都必须有相同数量的输出列; 而最终的结果集里的输出列的名字和类型要由第一个SELECT语句所选定的数据列的名字和类型来决定。如果对应于同一个输出列的数据列的类型不匹配, MySQL将自动地对来自第2个及后续数据表的数据行进行隐含的类型转换。

在默认的情况下, UNION将自动清除重复的数据行。这与使用了DISTINCT关键字的SELECT语句效果相同, 只是范围扩大到了所有的结果集上而已。UNION ALL保留重复的数据行, 即返回所有的数据行。

UNION允许在相关的SELECT语句里使用ORDER BY或LIMIT子句, 但必须把这条SELECT语句放在一对括号里。出现在整个UNION构造的末尾且没有用括号括起来的ORDER BY或LIMIT子句将作用于最终得到的UNION结果集——此时, ORDER BY子句所列举的输出

列将对应于第一个SELECT语句所选取的数据列。

UNION最早出现于MySQL 4.0.0版本。

#### D.1.67 UNLOCK TABLE

语法: UNLOCK { TABLE | TABLES }

这条语句将释放当前客户（程序）在各有关数据表上所持有的全部数据锁。

#### D.1.68 UPDATE

语法: UPDATE [LOW\_PRIORITY] [IGNORE] tbl\_name  
SET col\_name=expr [, col\_name=expr ] ...  
[WHERE where\_expr] [ORDER BY ... ] [LIMIT n]

UPDATE [LOW\_PRIORITY] [IGNORE] tbl\_name , tbl\_name ...  
SET col\_name=expr [, col\_name=expr ] ...  
[WHERE where\_expr] [ORDER BY ... ] [LIMIT n]

UPDATE语句的第一种语法形式用来修改数据表tbl\_name里的现有数据行，即对WHERE子句给出的表达式所选取的数据行进行修改。对于那些被选中的数据行，UPDATE语句的SET子句将把它们的各有关数据列设置为相应的表达式的值。如下所示：

```
UPDATE member SET expiration = NULL, phone = '197-602-4832'
WHERE member_id = 14;
```

如果没有给出WHERE子句，数据表中的所有记录都将被修改！

UPDATE语句的返回值是它实际修改的数据行的个数。但是，如果某个数据行的数据列没有真正地发生变化，该数据行就不会被统计在UPDATE语句的返回值里。换句话说，把某个数据列设置为它现有的值将被认为是对该数据行没有影响。如果你想知道的是到底有多少个数据行与UPDATE语句的WHERE子句相匹配而不是UPDATE语句实际改变了多少个数据行，就需要在建立与MySQL服务器的连接时对CLIENT\_FOUND\_ROWS标志进行设定；具体做法请参见附录F对mysql\_real\_connect()函数的介绍。

LOW\_PRIORITY子句将使UPDATE语句被延缓到没有客户（程序）在对给定数据表进行读操作时才执行。LOW\_PRIORITY子句最早出现于MySQL 3.22.5版本。

如果UPDATE语句对某个数据行的修改将导致数据表的某个惟一化索引出现重复的键值，这条UPDATE语句将报告出错并中止执行。但如果还给出了IGNORE子句，UPDATE语句就不会对这类数据行做出修改，也不会报告出错和中止执行。IGNORE子句最早出现于MySQL 3.23.16版本。

如果在UPDATE语句里还给出了ORDER BY子句，它将先对结果集进行排序，然后再按顺序对各有关数据行进行修改。UPDATE语句的ORDER BY子句最早出现于MySQL 4.0.0版本，它与SELECT语句中的ORDER BY子句有着同样的语法。

如果在UPDATE语句里还给出了LIMIT子句，它将只对结果集里的前n个数据行进行修改（如果结果集里的数据行不足n个，则修改所有数据行）。LIMIT子句最早出现于MySQL 3.23.3版本。

UPDATE语句的第二种语法形式与它的第一种语法形式用途相同，但它允许你同时给出多

个数据表，即它对数据行的修改操作将涉及到多个数据表。此时，可以在WHERE子句里给出一些对各有关数据表进行关联检索的筛选条件，再通过SET子句对各有关数据表的各有关数据列做出修改。比如说，下面这条语句将对数据表t1中的那些id值与数据表t2中的id值相匹配的数据行做出修改，把数据表t2中的quantity值拷贝到数据表t1里来：

```
UPDATE t1, t2 SET t1.quantity = t2.quantity WHERE t1.id = t2.id;
```

涉及多数据表的UPDATE机制最早出现于MySQL 4.0.2版本。

#### D.1.69 USE

语法：USE db\_name

把数据库db\_name选定为当前的默认数据库。此后，以tbl\_name形式（相对于database\_name.tbl\_name形式）给出的数据表就将是默认数据库里的数据表。如果该数据库不存在或者你没有该数据库的访问权限，USE语句将报告出错。

### D.2 SQL变量

从3.23.6版本开始，MySQL允许你定义自己的SQL变量并对它们进行赋值，还允许你在后续的其他语句里使用这些变量。

用户定义的SQL变量名必须以字符“@”开头，然后是一个由当前字符集里的字母、数字以及下划线（\_）、美元符号（\$）和小数点（.）等字符构成的名字。变量名区分字母的大小写情况。

SET语句允许你用“=”或“:=”操作符对变量进行赋值；但其他语句（比如SELECT语句）只允许你用“:=”操作符对变量进行赋值。可以在同一条语句里对多个变量进行赋值。如下所示：

```
mysql> SET @x = 0, @y = 2;
mysql> SET @color := 'red', @size := 'large';
mysql> SELECT @x, @y, @color, @size;
+-----+-----+-----+-----+
| @x    | @y    | @color | @size |
+-----+-----+-----+-----+
| 0     | 2     | red    | large |
+-----+-----+-----+-----+
mysql> SELECT @count := COUNT(*) FROM president;
+-----+
| @count := COUNT(*) |
+-----+
| 42                  |
+-----+
```

可以直接把数值、字符串或NULL值赋值给变量，也可以通过任意形式的表达式对变量进行赋值，而表达式里还允许出现其他变量。

在明确地对变量做出赋值之前，它们的值将是NULL。这些值只在本次会话过程中有效，当与MySQL服务器的连接被断开时，这些值也就不复存在了。

在返回多个数据行的SELECT语句里，变量的赋值操作将依次在每一个数据行上进行。最后那个值就是你在最后一个数据行上赋值的值。

在MySQL 4.1及以后的版本里，变量本身的字符集将与它们得到的赋值的字符集相同，如下所示：

```
mysql> SET @s = _latin1_de 'abc'; SELECT CHARSET(@s);
+-----+
| CHARSET(@s) |
+-----+
| latin1_de   |
+-----+
```

### D.3 注释语法

本节将向大家介绍如何在SQL代码里写出注释。它还指出了mysql客户程序在注释处理方面的一个缺陷——因为注释通常用在由mysql客户程序以批处理方式执行的查询文件里，所以大家在编写这类文件的时候尤其要注意这个问题。

MySQL服务器能够识别以下三种形式的注释：

- 从字符“#”开始一直到行尾的所有文字都将被视为注释内容。这种语法与大多数shell和Perl语言所使用的注释语法相同。
- “/\*”和“\*/”之间的所有文字都将被视为注释内容。这种形式的注释允许跨越多行。这种语法与C语言所使用的注释语法相同。
- 从3.23.3版本开始，MySQL还允许你以“-- ”（两个连字符加一个空格）来开始一个注释；从“-- ”到行尾的所有文字都将被视为注释内容。这种形式的注释多见于其他一些数据库软件，但那些数据库软件往往允许省略“-- ”中的空格。MySQL要求必须加上一个空格的原因是为了避免出现二义性问题，即确保value1 - value2形式的表达式即使在value2是一个负值的情况下也不会被错误地解释为一条注释。

在执行查询命令的时候，MySQL服务器会忽略语句中的注释，但以“/\*!”开始的C语言型注释却是个例外。这种注释的文字要求是一些SQL关键字，MySQL服务器会把这些关键字视为有关语句的组成部分之一。比如说，下面两条语句对MySQL服务器来说是等价的：

```
INSERT LOW_PRIORITY INTO mytbl SET ... ;
INSERT /*! LOW_PRIORITY */ INTO mytbl SET ... ;
```

这种形式的注释是为MySQL独有的扩展模块和关键字而准备的。因为MySQL能够识别出其中的关键字，而其他的SQL服务器却会忽略它们，所以我们就能写出一些这样的查询命令来：在MySQL数据库系统上，它们能利用MySQL所独有的某些功能；在其他的数据库系统上，它们也能毫无问题地工作。“/\*!”形式的注释最早出现于MySQL 3.22.7版本。

从MySQL 3.22.26版本开始，还可以通过在“/\*!”序列的后面加上一个版本号的办法来达到这样一个效果：如果你的MySQL服务器低于该注释所给出的版本号，它将忽略这条注释。请看下面这条语句：如果你的MySQL服务器不是3.23.3或者更高的版本，它就会忽略其中的注释：

```
UPDATE mytbl SET mycol = 100 WHERE mycol < 100 /*!32303 LIMIT 100 */;
```

与MySQL服务器相比，mysql客户程序的注释处理能力要弱很多。mysql客户程序用的语法



分析器没有MySQL服务器用的那么高级，所以它在遇到某些特定形式的C语言型注释时会受到愚弄。比如说，如果C语言型注释里有一个引号字符，mysql就会错误地认为自己是在分析一个字符串——它将去寻找这个字符串的末尾直到再遇到一个配对的引号字符为止。下面这两条语句就演示了这种情况：

```
mysql> SELECT /* I have no quote */ 1;
+----+
| 1 |
+----+
| 1 |
+----+
mysql> SELECT /* I've got a quote */ 1;
'>
```

对于上面的第一条语句，mysql客户程序顺利地完成了分析工作，把语句发送给MySQL服务器去执行，然后重新显示了一个“mysql>”提示符。但它在第二条语句上遇到了麻烦：这条语句中的注释里有一个不配对的引号字符，mysql客户程序因此而进入了字符串分析模式并在你按下回车键输入了这一行之后仍陷在该模式里，所以它显示了一个“>”提示符。脱离这种困境的办法是先敲入一个配对的引号字符、再敲入“\c”命令来取消这个查询。（附录E对mysql客户程序各种提示符的含义做了说明。）

出现在C语言型注释中的分号(;)字符也会让mysql受到愚弄。



## 附录E MySQL程序使用指南

本附录将对下列MySQL程序进行介绍。后面的内容将依次介绍每一个程序的用途、调用语法、所支持的选项以及所具有的内部变量。（这些程序将按它们名字的字母顺序排列先后，排名时不考虑“\_”或“.”字符。）如无特别注明，本附录所列举的选项和变量至少从MySQL 3.22.0版本开始就已经存在于MySQL软件中了。

- **libmysqld** 嵌入式MySQL服务器。它实际上并不是一个能够独立运行的程序，而是一个函数库，把它链接到其他程序里，就可以开发出自带MySQL服务器的应用软件来。
  - **myisamchk**和**isamchk** 两个用来检查和修复数据表、分析键字的分布情况、禁用或启用数据表索引的工具程序。
  - **myisampack**和**pack\_isam** 这两个工具程序用来对数据表进行压缩并生成只读的数据表。
  - **mysql** 一个具备命令行编辑功能、用来向MySQL服务器发送数据库查询命令的交互式客户端程序，也可以用它来以批处理方式执行存放在文件里的数据库查询命令。
  - **mysqlaccess** 一个用来测试数据库访问权限的脚本。
  - **mysqladmin** 一个用来执行各种系统维护和管理工作的工具程序。
  - **mysqlbinlog** 一个以ASCII格式显示二进制变更日志内容的工具程序。
  - **mysqlbug** 一个用来生成程序漏洞报告的脚本。
  - **mysqlcheck** 一个用来对数据表进行检查、修复、优化和分析的工具程序。
  - **mysql\_config** 当准备编译基于MySQL的程序时，可以利用这个工具程序来确定该程序的编译选项和标志。
  - **mysqld** MySQL服务器。只有先运行了这个程序，才能通过各种MySQL客户程序来访问那些受它管理的数据库。
  - **mysqld\_multi** 一个用来同时启动和关闭多个MySQL服务器的脚本。
  - **mysqld\_safe** 一个用来启动和监控MySQL服务器的脚本。（在MySQL 4之前的版本里，这个脚本的名字是**safe\_mysqld**。）
  - **mysqldump** 一个用来导出数据表内容的工具程序。
  - **mysqlhotcopy** 数据库备份工具程序。
  - **mysqlimport** 一个用来往数据表里批量加载数据的工具程序。
  - **mysql\_install\_db** 一个用来对MySQL服务器的数据目录和各种权限表进行初始化的脚本。
  - **mysql.server** 一个用来启动和关闭MySQL服务器的脚本。
  - **mysqlshow** 一个用来查看关于数据库或数据表的信息的工具程序。
- 在后面的内容里，将把语法说明中的可选信息放在方括号（[]）里。

## E.1 设置程序选项

大多数MySQL程序都有一些能够影响其操作行为的选项。这些选项既可以在命令行上给出，也可以在选项文件里给出。此外，还有一些选项可以通过设置相应的环境变量来进行设定。在命令行上给出的选项优先于以其他方式给出的选项；在选项文件里给出的选项又优先于通过环境变量设置的选项。

大多数MySQL程序都有一个名为--help的选项，这个选项的用途是把程序自带的在线帮助信息显示给用户。比如说，如果你拿不准应该如何使用mysqlimport程序，可以用下面这条命令来学习它的用法：

```
% mysqlimport --help
```

-?选项与--help的作用是一样的，但你的shell可能会把“?”字符解释为一个文件名通配符。

```
% mysqlimport -?
```

```
mysqlimport: No match.
```

如果遇到这种情况，请试试下面这条命令：

```
% mysqlimport -\?
```

有些选项只有在某种特定的条件下才会被列举在帮助信息里。比如说，--debug选项或与SSL相关的选项就只有在MySQL事先用调试机制或SSL支持机制编译的情况下才会出现在帮助信息里；而Windows系统所独有的选项（比如--pipe）也只有在Windows系统上才会出现在帮助信息里。

大多数选项都有一个长（完整单词）格式和一个短（单字符）格式。刚才描述的--help和-?就是一个典型的例子。后面跟有设置值的长格式选项要以--name = val的格式给出，其中的name是选项的名字，val是该选项的设置值。在大多数场合，如果短格式选项的后面还跟有一个设置值，选项和设置值之间允许出现空白字符。比如说，当给出一个用户名时，-usampadm和-u sampadm是等价的。-p（口令）选项是个例外——“-p”和它后面的口令值之间不允许有任何间隔。

在介绍每一个程序的时候，我们将把它目前所支持的所有选项全都列举出来。如果你的某个MySQL程序识别不出我们在本附录的相关条目里列举的某个选项，则很可能是因为这个程序的版本早于那个选项最早出现的版本。（请先核对一下有关的语法以确认你自己没有敲错那个选项。）

### E.1.1 MySQL 4.0.2——选项处理机制的分水岭

从MySQL 4.0.2版本开始，MySQL程序的选项处理机制发生了重大的改变，新机制为布尔型选项（即那些设置值为开/关形式的选项）提供了一种更为统一的设置格式。这类选项现在有一个基本形式和一组标准的相关形式，如下表所示：

选 项	含 义
<code>--name</code>	基本形式；激活该选项
<code>--enable-name</code>	<code>--enable-</code> 前缀；激活该选项
<code>--disable-name</code>	<code>--disable-</code> 前缀；禁用该选项
<code>--skip-name</code>	<code>--skip-</code> 前缀；禁用该选项
<code>--name = 1</code>	<code>= 1</code> 后缀；激活该选项
<code>--name = 0</code>	<code>= 0</code> 后缀；禁用该选项

比如说，许多MySQL命令都支持用来激活客户/服务器协议中的压缩功能的`--compress`选项。在MySQL 4.0.2之前的版本里，如果给出了这个选项，则激活压缩功能；如果没有给出这个选项，则禁用压缩功能。现在仍可以这样做，但在MySQL 4.0.2及以后的版本里，`--enable-compress`和`--compress=1`也将被解释为激活压缩功能，而`--disable-compress`、`--skip-compress`和`--compress=0`将被解释为不使用压缩功能。

稍后，在本附录依次介绍各程序用法的时候，我们将给支持上述解释办法的选项（即允许用上表中的以前缀、后缀形式给出的选项）加上一个“（布尔）”标记。带有这个标记的选项现在的推荐用法是以`--enable-`、`--disable-`和`--skip-`前缀格式来给出。但在运行那些版本早于MySQL 4.0.2的老程序时，可能需要以`=1`或`=0`后缀格式甚至是另外一种语法来给出有关的选项。比如说，老版本的mysql程序中有一个`--no-named-columns`选项，这个选项的作用是让MySQL不显示查询结果集里的输出列标题。在MySQL 4.0.2及以后的版本里，这个选项的基本形式是`--named-columns`（这也是该选项的默认设置）；如果你不想在输出报告里看到输出列的标题，就需要明确地给出`--disable-named-columns`或`--skip-named-columns`选项。

因为MySQL 4.0.2版本之前的代码在新版MySQL软件里仍占有相当大的比例，所以你在运行某个程序的时候很可能会拿不准到底应该如何给出有关的选项。（大多数选项在各种MySQL版本下都是一样的，但有些布尔型选项的语法却发生了比较大的变化。）这个问题将随着时间的推移以及MySQL 4逐步取代各种老版本而逐渐被消除。当你拿不准应该如何给出某个程序的选项时，只需用`--help`选项来查看它所支持的选项格式。

MySQL 4.0.2版本在选项处理机制方面的改进还包括以下几个方面：

- 选项名允许以最短的无二义前缀形式给出，这为那些名字很长的选项的设定工作提供了方便。
- 允许你通过将变量名当做选项名来从命令行或者在选项文件里设置程序变量。也就是说，不必再通过`--set-variable`或者`-O`选项来设置有关的变量。这方面的详细讨论请参见本附录后面内容里的“设置程序变量”小节。
- `mysqld`程序还支持使用`--maximum-`前缀来给MySQL程序中的用户定义变量设置一个最大值。比如说，MySQL服务器允许用户通过改变`sort_buffer_size`变量值的办法来调整排序缓冲区的尺寸。如果想把这个变量的最大值设置为64MB，就需要在启动MySQL服务器的时候给出一个`--maximum-sort_buffer_size=64MB`选项。
- 新增了一个`--loose-`前缀，它能使来自不同版本的MySQL程序对选项的格式不那么挑剔。比如说，4.1及更高版本的MySQL服务器都能识别`--old-passwords`选项，但老版本的服务器

就不行了。如果把这个选项以--loose-old-passwords的形式给出,那么,4.0.2及更高版本的MySQL服务器就能根据它自己是否支持--old-passwords选项而使用或者忽略这个选项。

### E.1.2 MySQL程序的标准选项

有些选项在各种MySQL程序里都有着同样的含义和作用,我们把这些选项称为MySQL程序的“标准”选项。为简洁起见,在这里对它们做一次全面的介绍,等介绍到某个具体的程序时,将只在“所支持的标准选项”小节里列出该程序所支持的标准选项而不再重复介绍它们的用途和用法了。(本小节只列出了有关选项的长格式名称;如无特别说明,完全可以使用相应的短格式选项来运行有关的MySQL程序。)

下面就是这些标准选项的一份清单:

- --character-sets-dir = *dir\_name* 用来存放字符集文件的目录。
- --compress 或 -C (布尔) 使用客户/服务器通信协议中的压缩功能——如果服务器支持的话。这个选项只能由客户程序使用。它最早出现于MySQL 3.22.3版本。
- --debug = *debug\_options* 或 -# *debug\_options* 打开调试输出。如果MySQL在编译时没有激活调试支持机制,这个选项将不可用。*debug\_options*是由一个或者多个以冒号(:)分隔的选项构成的字符串。它的典型设置值是d:t:o, *file-name*——即激活调试功能、打开进入和退出函数调用时的跟踪机制、并把输出发送到文件*file-name*里去。  
MySQL源代码发行版本里的dbug/dbug.c文件对各种可供选用的调试选项进行了说明。如果你想做更多的调试,请自行查阅这个文件。
- --default-character-set = *charset* 默认字符集的名字。
- --help 或 -? 显示帮助信息并退出。
- --host = *host\_name* 或 -h *host\_name* 将要连接的主机(即MySQL服务器在其上运行的主机)。这个选项只能由客户程序使用。
- --password [= *pass\_val*] 或 -p[*pass\_val*] 用来连接MySQL服务器的口令。如果你没有在这个选项名的后面给出*pass\_val*,程序将提示你输入之。如果你给出了*pass\_val*,它必须紧跟在选项名的后面,中间不留任何空隙。(也就是说,这个选项的短格式必须写成-p *pass\_val*,不能写成-p *pass\_val*。)这个选项只能由客户程序使用。
- --pipe 或 -W 使用一个命名管道来连接MySQL服务器。这个选项只能由运行在Windows系统中的客户程序使用,而且只能用来连接那些支持使用命名管道的基于Windows NT的服务器。
- --port = *port\_num* 或 -P *port\_num* 对于客户程序,这是它连接MySQL服务器时使用的端口号。这个选项建立的是一条TCP/IP连接,TCP/IP连接中的服务器主机不能是localhost(如果你使用的是UNIX系统的话)或“.”(如果你使用的是Windows系统的话)。对于mysqld,这个选项指定的是它将在其上监听TCP/IP连接的端口。
- --set-variable *var=value* 或 -O *var=value* 这个选项允许你对程序操作参数进行赋值。*var*是变量名,*value*是该变量的赋值。



MySQL 4.0.2及以后的版本允许直接使用变量的名字来进行赋值，所以这个选项在新版本里已经不再是必要的了。后面内容里的“设置程序变量”小节对新、旧两种变量赋值语法做了详细的介绍。

- `--silent` 或 `-s` 在沉默模式下运行。这并不意味着程序将完全沉默，但程序在这种模式下产生的输出信息的确要比正常情况时的少。有些程序允许重复多次地给出这个选项，这将使程序更加沉默。（这也适用于选项文件。）
- `--socket = file_name` 或 `-S file_name` 对于UNIX系统上的客户程序，这是它在连接主机localhost上的MySQL服务器时使用的套接字文件的路径名。对于Windows系统上的客户程序，这是它在以主机名“.”来连接MySQL服务器时使用的命名管道的名字。
- `--user = user_name` 或 `-u user_name` 对于客户程序，这是它在连接MySQL服务器时使用的MySQL用户名。如果你没有给出这个选项，它的默认值将是你的登录名（如果你使用的是UNIX系统的话）或ODBC（如果你使用的是Windows系统的话）。对于mysqld，这个选项指定了用来运行MySQL服务器的UNIX账户名。不过，要想让这个选项有效果，MySQL服务器必须是以根用户root身份启动才行，因为只有这样才能改变自己的用户ID。
- `--verbose` 或 `-v` 在详细信息模式里运行，程序将比正常情况产生更多的输出。有些程序允许你重复多次地给出这个选项，这将使程序产生更多的输出信息。（这也适用于选项文件。）
- `--version` 或 `-V` 让程序显示其版本信息字符串并退出。

### 1. SSL标准选项

下列选项用来建立安全化连接。它们最早出现于MySQL 4.0.0版本，但只有在MySQL软件里编译有SSL支持机制时才能使用。与建立安全化连接有关的详细讨论见本书第12章。

- `--ssl` 允许SSL连接（如果你给出的是`--disable-ssl`选项，则将禁用之）。与SSL有关的其他选项都隐含着`--ssl`选项功能。
- `--ssl-ca = file_name` 颁证机构的证书文件的路径名。
- `--ssl-capath = dir_name` 用来保存信任证书的目录的路径名，用于证书验证工作。
- `--ssl-cert = file_name` 证书文件的路径名。
- `--ssl-cipher = str` 这个字符串给出的是用来对客户/服务器之间的通信进行加密的SSL加密类型的名称。这个字符串应该给出一个或者多个以逗号分隔的加密类型的名称。
- `--ssl-key = file_name` 密钥文件的路径名。

### 2. 设置程序变量

部分MySQL程序有一些允许由用户进行设置的变量（操作参数）。从MySQL 4.0.2版本开始，可以像对待程序选项（即把变量名视为选项名）那样来设置有关变量的值。比如说，如果想在启动mysql程序时把`connect_timeout`变量的值设置为10，可以使用如下所示的命令：

```
% mysql --connect_timeout=10
```

这种语法还允许你把变量名中的下划线字符（`_`）写成连字符（`-`），这就使变量选项与程序选项更相似了：

```
% mysql --connect-timeout=10
```

MySQL 4.0.2之前的版本要求使用--set-variable选项（或者它的短格式-O）来设置变量。如下所示：

```
% mysql --set-variable=connect_timeout=10
```

```
% mysql -O connect_timeout=10
```

MySQL 4.0.2及以后的版本仍支持--set-variable和-O选项的使用，但并不推荐这样做。

对于代表缓冲区尺寸或者各种长度的变量值，如果不带计量单位后缀，则按字节计算；如果带有“K”或“M”后缀，则分别按千字节（KB）和兆字节（MB）计算。计量单位后缀不区分字母的大小写情况；“k”或“m”与“K”或“M”是等价的。从MySQL 4.0.2版本开始，还可以用“g”或“G”来代表千兆字节（GB）。

在本附录后面的内容里，我们将在介绍各个程序的时候把与之有关的变量也列举出来。在使用--help选项执行某个程序而显示出来的帮助信息里也能看到与该程序有关的变量。

### E.1.3 选项文件

选项文件最早出现于MySQL 3.22.10版本，大多数MySQL程序都支持选项文件的使用。作为一种保存程序选项的手段，选项文件使我们不必在执行某个程序时每次都在命令行上敲入一大堆的选项。可以在MySQL软件主安装目录的share/mysql目录或者MySQL源代码发行版本的support-files目录里找到一些示范性的选项文件。

在选项文件里设定的选项其优先级低于在命令行上明确给出的选项。如果在命令行上设定的选项值与你事先在选项文件里设定的选项值不一样，程序将使用命令行上给出的选项值去进行有关的操作。

支持选项文件的MySQL程序将顺序到几个地方去寻找选项，但选项文件不存在却不会被视为是一个错误。在UNIX系统上，MySQL程序将依次对以下几个文件里的选项进行处理：

文 件 名	内 容
/etc/my.cnf	全局级选项
DATADIR/my.cnf	服务器级选项文件；与特定MySQL服务器有关的选项
~/my.cnf	用户级选项文件；与特定用户有关的选项

在Windows系统上，MySQL程序将依次对以下几个文件里的选项进行处理：

文 件 名	内 容
SYSTEMDIR/my.ini	全局级选项
C:\my.cnf	全局级选项
DATADIR\my.cnf	服务器级选项文件；与特定MySQL服务器有关的选项

DATADIR代表的是你机器上的MySQL数据目录的路径名。（这个路径名已经被编译到MySQL服务器里了，不能通过--datadir选项来改变它。）在Windows系统上，DATADIR是C:\mysql\data。

SYSTEMDIR代表的是Windows系统目录的路径名（它通常是C:\Windows或C:\WinNT）。

只要是会用到选项文件的MySQL程序，就都要用到全局级选项文件；而保存在服务器数据目录里的选项文件只有那些以该目录作为默认数据目录的程序才会用到。用户级选项文件则只有由该用户运行的程序才会用到。

Windows用户在使用选项文件的时候还需要注意以下几个问题：

- Windows路径通常包含有反斜线字符（\），可这个字符在MySQL里是被当做转义字符来使用的。因此，必须把取值为路径名的选项里的反斜线字符写成斜线字符（/）或双反斜线字符（\\）。
- Windows往往会把文件名中的扩展名隐藏起来。如果你创建了一个名为my.cnf的选项文件，Windows可能会把它显示为my。如果你发现了这个“错误”并在Windows的资源管理器里把这个文件重新命名为my.cnf，就会发现它不再起作用了——因为你实际上把它的名字从my.cnf重新命名为my.cnf.cnf了！
- 选项文件必须是普通的文本文件。可以用字处理软件来编写选项文件，但必须把它保存为文本格式而不能把它保存为字处理软件专用的文档格式。

有四个与选项文件的处理工作有关的选项是大多数MySQL程序都能识别和支持的，它们的含义如下所示；只要你打算使用它们当中的任何一个，就必须把它写成命令行上的第一个选项。

- `--defaults-extra-file = file_name` 除标准的选项文件外，MySQL程序还必须从这个文件里读取选项。MySQL程序将在读完全局级和服务级选项文件之后、在读取用户级选项文件之前去读取这个文件里的选项。这个选项最早出现于MySQL 3.23.26版本，但safe\_mysqld和mysql\_install\_db脚本对这个选项的支持却是分别从3.23.27和3.23.29版本才开始的。
- `--defaults-file = file_name` 只从这个文件里读取选项。在默认的情况下，MySQL程序会依次到几个地方去寻找选项文件，但如果给出了--defaults-file选项，它将只读取指定文件里的选项。这个选项最早出现于MySQL 3.22.23版本。
- `--no-defaults` 禁止使用任何选项文件。此外，这个选项还会导致--defaults-file等与选项文件有关的其他选项失去作用。
- `--print-defaults` 因为有选项文件和环境变量的缘故，所以即使没有在命令行上给出任何选项，MySQL程序也会按照一些“默认的”选项设置去执行。--print-defaults将把这些“默认的”选项（来自选项文件和环境变量的选项设置情况都包括在内）显示出来。可以利用这个选项来检查某个选项文件的设置情况是否正确。此外，如果MySQL程序的行为看起来像是使用了一个你本人从没给出过的选项，就应该使用--print-defaults选项来检查一下，看它是不是来自某个选项文件。

如果用--help选项去执行某个程序，就可以在帮助信息中看到该程序通常会去读取的选项文件的清单。（这份清单会受到--defaults-file、--defaults-extra-file、--no-defaults等选项的影响。）

选项是分组给出的。下面是一个示例：

```
[client]
user=sampadm
password=secret

[mysql]
no-auto-rehash

[mysqlshow]
status
```

选项组的名字必须写在方括号里。`[client]`是一个特殊的组名，这一组里的选项将作用于所有的客户程序。其他的组名通常都对应于某个具体的客户程序。在上面的例子里，组名`[mysql]`表示该组选项是供mysql客户程序使用的，组名`[mysqlshow]`则表示该组选项是供mysqlshow客户程序使用的。标准的MySQL客户程序会到`[client]`选项组和与它同名的选项组里去查找选项。比如说，mysql会到`[client]`和`[mysql]`选项组里去查找选项，mysqlshow则会到`[client]`和`[mysqlshow]`里去查找选项。

跟在某一个组名后面出现的选项都与该选项组相关联。一个选项文件可以包含任意多个选项组，后出现的选项组将优先于先出现的选项组。如果某个选项在某个程序将会查找的多个选项组里都出现，该程序最终将使用这个选项最后一次出现时的设置值。

在选项文件里，每个选项都独占一行。每行的第一个单词是该选项的名字，它必须以不带前导连字符的长格式形式写出。（比如说，在命令行上，可以用`-C`或`--compress`来设置压缩功能；但在选项文件里，只能使用`compress`。）只要是MySQL程序支持的长格式选项，就可以写到一个选项文件里去。选项和选项值（如果有的话）要用等号（`=`）隔开。

请看下面这条命令行命令：

```
% mysql --compress --user=sampadm --set-variable=max_allowed_packet=16M
```

如果你想用选项文件里的`[mysql]`组来给出同样的设置信息，可以像下面这样做：

```
[mysql]
compress
user=sampadm
set-variable=max_allowed_packet=16M
```

注意，在选项文件里，`--set-variable`后面紧跟着等号（`=`）字符，在变量名和变量值之间还有一个等号字符。（在命令行上，可以用空格来代替“`=`”字符。）

在MySQL 4.0.2及以后的版本里，还可以把变量名当做选项那样来对待，如下所示：

```
% mysql --compress --user=sampadm --max_allowed_packet=16M
```

或者：

```
[mysql]
compress
user=sampadm
max_allowed_packet=16M
```

选项文件里的空白行和以“`#`”或“`;`”开始的行将被视为注释而不做处理。选项设置行上的前导空格（如果有的话）将被忽略。

在选项文件里，某些特殊字符需要用转义序列来表示（如下表所示）：

转义序列	含 义
\b	退格符
\n	换行符
\r	回车符
\s	空格
\t	制表符
\\	反斜线字符

千万不要把仅有某一个MySQL客户程序能理解的选项放到[client]选项组里去。比如说，如果把mysql独有的skip-line-numbers选项放到了[client]选项组里，你将突然发现其他客户程序，比如mysqlimport，都不能工作了。（你将看到一条出错信息，后面还有一段帮助信息。）正确的做法是把skip-line-numbers选项安排在[mysql]选项组里。

下列选项是全体或大多数MySQL程序都能识别和处理的，把它们放到[client]选项组里一般不会造成什么问题：

```
character-sets-dir=charset_directory_path
compress
connect-timeout=seconds
database=db_name
debug
default-character-set=charset_name
disable-local-infile
host=host_name
init-command=query
interactive-timeout=seconds
local-infile
password=your_pass
pipe
port=port_num
return-found-rows
socket=socket_name
ssl-ca=ssl_certificate_authority_file
ssl-capath=ssl_certificate_authority_directory_path
ssl-cert=ssl_certificate_file
ssl-key=ssl_key_file
timeout=seconds
user=user_name
```

选项文件的支持机制最早出现于MySQL 3.22.10版本，但有些选项是后来又新增的：

- debug选项最早出现于MySQL 3.22.11版本。
- return-found-rows选项最早出现于MySQL 3.22.15版本。
- character-sets-dir和default-character-set选项最早出现于MySQL 3.23.14版本。
- interactive-timeout选项最早出现于MySQL 3.23.28版本。
- connect-timeout选项最早出现于MySQL 3.23.49版本。connect-timeout是现在已不推荐使用的timeout选项的同义词。



- `disable-local-infile`和`local-infile`选项最早出现于MySQL 3.23.49版本。
- `ssl-ca`、`ssl-capath`、`ssl-cert`、`ssl-key`等选项最早出现于MySQL 4.0.0版本。

#### 用my\_print\_defaults来检查选项

如果你想知道某个MySQL程序都使用了哪些选项，可以用`my_print_defaults`工具来查看之。这个工具将从选项文件里把与该程序有关的选项都查找并显示出来。比如说，`mysql`程序要使用来自`[client]`和`[mysql]`组的选项。如果你想知道选项文件里都有哪些会影响到`mysql`程序的选项，就要发出一条下面这样的命令：

```
% my_print_defaults client mysql
```

再比如说，服务器程序`mysqld`要使用来自`[mysqld]`和`[server]`组的选项。如果你想知道这两个选项组都列举了哪些选项，就要使用下面这样的命令：

```
% my_print_defaults mysqld server
```

UNIX系统下的`my_print_defaults`工具最早出现于MySQL 3.23.19版本，Windows系统下的这个工具最早出现于MySQL 4.0.4版本。

#### 让用户级选项文件做到专人专用

在UNIX系统上，先设置好文件的属主，再把文件的访问模式设置为600或400，其他用户就不能读出它的内容了。利用这一点，我们就能让用户级选项文件做到专人专用。比如说，谁都不想让自己的MySQL用户名和口令信息被其他用户偷看到。如果想让你自己的选项文件只能由你本人来读取的话，就需要在主目录里发出下面这样的命令：

```
% chmod 600 .my.cnf
% chmod go-rwx .my.cnf
```

#### E.1.4 环境变量

MySQL程序会去检查一些环境变量以获得选项设置。环境变量的优先程度是很低的，在选项文件里或者在命令行上设置的选项都优先于用环境变量设置的选项。

MySQL程序检查以下环境变量：

- `MYSQL_DEBUG` 调试时使用的选项。如果在编译MySQL软件时没有激活调试支持机制，这个变量将没有任何效果。设置`MYSQL_DEBUG`变量与使用`--debug`选项的情况相同。
- `MYSQL_PWD` 用来连接MySQL服务器的口令。设置`MYSQL_PWD`变量与使用`--password`选项的情况相同。

用`MYSQL_PWD`变量来保存口令是不安全的，系统上的其他用户可以轻易地发现它的值。比如说，`ps`命令能把其他用户的环境变量设置显示出来。

- `MYSQL_TCP_PORT` 对于客户程序，这是它以TCP/IP方式连接MySQL服务器时使用的端口号。对于`mysqld`，这个选项指定的是它将在其上监听TCP/IP连接的端口。设置

MYSQL\_TCP\_PORT变量与使用--port选项的情况相同。

- **MYSQL\_UNIX\_PORT** 对于客户程序，这是它在连接主机localhost上的MySQL服务器时使用的套接字文件的路径名。对于mysqld，这是它在其上监听本地连接的套接字。设置MYSQL\_UNIX\_PORT变量与使用--socket选项的情况相同。
- **TMPDIR** 一个路径名，MySQL将把临时文件创建在这个目录里。设置这个变量与使用--tmpdir选项的情况相同。
- **USER** 用来连接MySQL服务器的MySQL用户名。这个选项只能由运行在Windows系统下的客户程序使用；设置这个变量与使用--user选项的情况相同。

mysql客户程序还要多检查三个环境变量：

- **MYSQL\_HISTFILE** 用来保存命令行历史的文件名。这个变量的默认值是\$HOME/.mysql\_history，\$HOME是主目录的位置。
- **MYSQL\_HOST** 将要连接的主机（即MySQL服务器在其上运行的主机）。设置这个变量与使用--host选项的情况相同。
- **MYSQL\_PS1** 代替“mysql>”而充当主提示符的字符串。这个字符串可能包含有我们将在mysql条目里介绍的特殊序列。

## E.2 libmysqld

libmysqld是一个嵌入式MySQL服务器。实际上它并不是一个能够独立运行的程序，而是一个函数库。把它链接到主机程序里，你就可以开发出自带MySQL服务器的应用软件来。与mysqld相比，libmysqld所支持的功能要少一些。比如说，为了节约空间，libmysqld禁用了ISAM支持；因为与主机程序的通信用不着网络，所以它干脆没把SSL支持组件包括在内；服务器既不能去连接其他的程序，也不能被其他程序连接——这使嵌入式服务器根本不能用在镜像机制中。

不过，因为嵌入式服务器不需要具备网络能力，也根本不需要生成任何输出文件，所以libmysqld非常适合用来创建将从光驱等只读介质直接运行的应用软件，或者用来创建将在独立运行的机器（比如自助式信息查询机）上使用的应用软件。

libmysqld本身是无法从命令行上读取选项的，但你可以安排其他程序把选项传递给它，它将像其他MySQL程序那样对选项做出处理。libmysqld能够识别和支持mysqld程序的大多数选项，详细情况请参考后面对mysqld程序的介绍。与服务器操作有关但libmysqld不支持的选项（比如与镜像机制有关的各种选项）将被忽略。

有关嵌入式服务器的详细讨论可以在本书的以下章节中找到：

- 第6章对内建有嵌入式MySQL服务器的主机程序的编写方法进行了探讨。
- 第11章对MySQL服务器的常规管理工作进行了介绍。
- 附录F对与嵌入式MySQL服务器有关的C语言函数进行了介绍。

## E.3 myisamchk和isamchk

这两个工具程序能够让你完成以下工作：检查和修复损坏了的数据表、查看关于数据表信

息、对索引键值的分布情况进行分析、禁用或激活索引。第4章对键值分析和索引禁用等问题做了比较详细的介绍。第13章对数据表的检查和修复工作做了比较详细的介绍。

`myisamchk`用来对MyISAM存储格式的数据表进行处理,这类数据表的数据和索引文件分别以`.MYD`和`.MYI`为文件扩展名。`isamchk`用来对ISAM存储格式的数据表进行处理,这类数据表的数据和索引文件分别以`.ISD`和`.ISM`为文件扩展名。如果你用`myisamchk`去处理ISAM数据表(或者用`isamchk`去处理MyISAM数据表),它将显示一条警告信息并忽略数据表。

在使用`myisamchk`和`isamchk`对某个数据表进行检查和修复期间,应该禁止MySQL对数据表的访问,具体做法可以在本书第13章内容里查到。

### E.3.1 用法

```
myisamchk [ options ] tbl_name[.MYI] . . .
isamchk [ options ] tbl_name[.ISM] . . .
```

如果没有给出任何选项,这两个程序将去检查给定数据表是否有错误。如果给出了一些选项,它们就将按照那些选项的含义去进行操作。如果你打算进行的操作会改变数据表的内容,应该先对各有关数据表进行备份。

`tbl_name`参数可以是一个数据表的名字,也可以是数据表的索引文件的名字。(MyISAM数据表的索引文件以`.MYI`为扩展名,ISAM数据表的索引文件以`.ISM`为扩展名。)使用索引文件名的好处是可以利用文件名通配符获得只用一条命令就对多个数据表进行处理的效果。比如说,如果想对某个目录里所有的MyISAM或ISAM数据表进行检查,只需发出下面这样的命令就行了:

```
% myisamchk *.MYI
% isamchk *.ISM
```

这两个程序并不要求有待检查的数据表必须存放在某个特定的地方。如果你想检查的数据表不在当前目录里,就必须给出它们所在的目录的路径名。因为这两个程序不要求数据表文件必须存放在服务器的数据目录里,所以可以先把它们拷贝到另外一个目录、然后再对那些副本文件而不是原始文件进行操作。

有些选项需要用到索引的序号。索引是从1开始编号的。可以用`SHOW INDEX`查询或者用`mysqlshow --keys`命令来查看某特定数据表各个索引的编号顺序;而`myisamchk`和`isamchk`将按照各有关索引在`Key_name`输出列里的先后顺序对它们进行检查和修复。

### E.3.2 `myisamchk`和`isamchk`都支持的标准选项

```
--character-sets-dir    --help                --verbose
--debug                 --set-variable       --version
--default-character-set --silent
```

`--default-character-set`选项只能由`iasmchk`程序使用。`--silent`选项意味着只显示出错信息,而`--verbose`选项在同时还给出了`--description`或`--extend-check`选项,或者同时还给出了`myisam`程序的`--check`选项的时候将显示更多的信息。在同一条命令里给出多个`--silent`或`--verbose`选项将加强它们的效果。

### E.3.3 myisamchk和isamchk都支持的其他选项

下面是一些myisamchk和isamchk都支持的“非标准”选项：

- `--analyze` 或 `-a` 进行键值分布分析。这可以帮助服务器加快基于索引的查找和关联操作的执行速度。完成分析工作之后，以`--description`加`--verbose`选项再次运行myisamchk或isamchk程序，就能查看到键值分布信息。
- `--block-search=n` 或 `-b n` 数据表的第 $n$ 个区块是从第几个数据行开始的。这是一个用于调试工作的选项。
- `--description` 或 `-d` 显示关于数据表的描述性信息。
- `--extend-check` 或 `-e` 对数据表做进一步的检查。需要用到这个选项的场合是非常少的，因为myisamchk和isamchk其他几个不这么高级的检查功能已经足以把所有的错误都查出来了。
- `--force` 或 `-f` 强制进行对数据表的检查或修复工作，即使在已经存在着一个对应于数据表的临时文件的情况下也如此进行。一般说来，如果myisamchk或isamchk发现已经存在着一个名为`tbl_name.TMD`的文件，它就会简单地在显示一条出错信息后退出执行，因为这通常意味着有另外一个myisamchk或isamchk程序实例正在运行。但也存在着这样一种可能性：你在这两个工具程序尚未执行完毕的时候强行终止了它们的运行——因为来不及把临时文件安全地删除掉，所以它们就遗留在了系统里。如果你知道自己的系统曾发生过这样的事情，就应该用`--force`选项来执行这两个工具程序，让它们不管是否存在临时文件都强行运行。（另一种做法是以手动方式去删除临时文件。）

如果在检查数据表的时候使用了`--force`选项，那么，只要myisamchk或isamchk发现某个数据表有问题，就会自动使用`--recover`选项重新启动自己。此外，从MySQL 4.0.2版本开始，myisamchk还将自动刷新数据表的状态——就好像还同时使用了`--update-state`选项那样。

- `--information` 或 `-i` 显示关于数据表内容的统计信息。
- `--keys-used=n` 或 `-k n` 这个选项要与`--recover`选项配合使用。对于isamchk，这个选项的作用是让MySQL只刷新前 $n$ 个索引。换句话说，这个选项将禁用编号大于 $n$ 的那些索引。对于myisamchk， $n$ 将被用做一个表明允许使用哪些索引的位掩码；第一个索引对应于第0位。在这两种场合里，把 $n$ 设置为0都意味着要禁用所有的索引。如果使用得当，这个选项将改善INSERT、DELETE、UPDATE等操作的性能。重新启用某个索引将恢复其正常的索引行为，具体做法是：对于isamchk，把 $n$ 设置为有关索引的最大编号值即可；对于myisamchk，把掩码 $n$ 中与各有关索引相对应的位全都设置为1即可。
- `--no-symlinks` 或 `-l` 在没有给出这一选项的情况下，如果某个`tbl_name`参数是一个符号链接，myisamchk或isamchk将去检查和修复符号链接所指向的数据表。但如果给出了这个选项，符号链接机制将不起作用——符号链接将被替换为有关文件被修复后的新版本。对于myisamchk，这个选项在MySQL 4及以后的版本里不可用——因为这些新版本里的myisamchk程序在修复数据表的过程中不删除符号链接。
- `--quick` 或 `-q`（布尔）与`--recover`选项配合使用，加快`--recover`选项对数据表的修复工作。给`--recover`选项加上一个`--quick`选项将不对数据文件进行修复。如果你想在发现数据表的



惟一化索引里有重复键值时对数据文件进行修复，就需要给--recover选项加上两个--quick选项。

- --recover 或 -r 对数据表进行常规的修复操作。这可以修复数据表的大多数错误，但不能消除惟一化索引中的键值重复现象。
- --safe-recover 或 -o 在确保安全的前提下对数据表进行修复。虽说比--recover方法执行得慢，但这个选项能够修复一些--recover选项不能修复的错误。此外，--safe-recover使用的磁盘空间也要少于--recover。
- --sort-index 或 -S 对索引块进行排序以加快此后的检索操作依次读取多个索引块的速度。
- --sort-records=*n* 或 -R *n* 根据数据记录在第*n*个索引中的先后顺序对它们进行排序，这将加快基于索引的检索操作的执行速度。在你第一次对数据表进行这种操作时，因为数据记录从没进行过排序，所以速度可能会很慢。如果你使用的是MySQL 3.23.28或更高的版本，就可以用ALTER TABLE ... ORDER BY语句来完成与--sort-records选项同样的工作，而且在速度上往往会更快一些。
- --unpack 或 -u 解压缩一个压缩文件。myisamchk能够对用myisampack程序压缩的MyISAM文件进行解压缩；isamchk能够对用pack\_isam程序压缩的ISAM文件进行解压缩。这个选项可以用来把压缩的只读数据表转换为可修改的格式。它不能与--quick或--sort-records选项一起使用。
- --wait 或 -w 如果数据表已被锁定，则等到该数据表可用为止。如果没有使用--wait选项，这两个程序在遇到被锁定的数据表时将等待10秒，如果到那时还没有获得数据锁的话将显示一条出错信息。

#### E.3.4 myisamchk独有的选项

虽然isamchk程序没有独有的选项，但myisamchk程序还是有几个独有的选项的：

- --backup 或 -B 在myisamchk程序其他选项会对数据表做出修改的情况下，这个选项将在修改发生之前先对数据表进行备份，备份文件的名称是tbl\_name-time.BAK。time是一个时间戳的数值表示形式。备份文件将被写到原始数据表所在的目录里。这个选项最早出现于MySQL 3.23.25版本，BACKUP TABLE语句也是从这个版本开始才出现的，但它们二者的功用并不相同。
- --check 或 -c 检查数据表中的错误。这是myisamchk程序在你没有给出任何选项时的默认动作。
- --check-only-changed 或 -C 只对上次检查后又发生过修改的数据表进行检查。这个选项最早出现于MySQL 3.23.22版本；在此之前，请使用--fast选项。
- --correct-checksum 对于使用了CHECKSUM = 1选项而创建出来的数据表，这个选项将确保该数据表里的校验和信息正确无误。这个选项最早出现于MySQL 4.0.0版本。
- --data-file-length=*n* 或 -D *n* 当数据文件的长度增长到MySQL软件本身或者操作系统所容许的上限，或者当数据表里的数据行的个数增长到MySQL内部数据结构所容许的上限时，



我们将无法再往里面添加新的数据记录。此时，需要重建这个数据文件并通过--data-file-length选项给新数据文件设定一个允许它增长到的最大长度。这个选项的设置值以字节为计量单位。这个选项必须与--recover或--safe-recover选项一起使用才会生效。

- --fast 或 -F 只对没有被正常关闭的数据表进行检查。在MySQL 3.23.22之前的版本里，这个选项的行为与--check-only-changed选项的相同，即只对上次检查后又发生过修改的数据表进行检查。
- --medium-check 或 -m 对数据表进行中级修复。它比--extend-check方法执行得快，但不如后者那么彻底（myisamchk程序的帮助信息说这个方法“只能找出99.99%的错误”）。这种检查模式对大多数场合来说都应该是足够的了。中级检查模式的工作原理是这样的：先把索引中的键值的CRC校验和计算出来，再把它们与根据数据文件中的被索引数据列而计算出来的CRC校验和进行比较。
- --parallel-recover 或 -p 像--recover选项那样对数据表进行修复，但将使用多个线程去并行地重建各有关索引。这要比以非并行方式来重建索引的速度更快，但这个选项目前仍被认为是试验性质的。这个选项最早出现于MySQL 4.0.2版本。
- --read-only 或 -T 不把数据表标记为已被检查过。
- --set-auto-increment[=*n*] 或 -A[*n*] 对AUTO\_INCREMENT计数器进行设置，此后的序列编号值将从*n*开始。如果数据表里已经存在有序列编号等于*n*的记录或者你没有给出*n*值，这个选项将把下一个AUTO\_INCREMENT值设置为“当前最大编号值+AUTO\_INCREMENT递增量”（这个递增量通常是1）。  
如果你使用的是这个选项的短格式，-A和*n*值之间不允许有空格；如果有空格的话，MySQL将无法对*n*值做出正确的解释。
- --set-character-set=*charset* 在重建索引的时候，使用给定字符集的排位顺序来确定各索引数据项的顺序。如果你改变了MySQL服务器的默认字符集，就应该使用这个选项来对MyISAM数据表里的索引重新进行排序。这个选项通常要与--recover和--quick选项一起使用。这个选项最早出现于MySQL 3.23.14版本。
- --sort-recover 或 -n 对数据表强制进行排序模式恢复，不管进行这种恢复所必需的临时文件会增大到什么程度。这个选项最早出现于MySQL 3.23.22版本。
- --start-check-pos=*n* 从位置*n*开始读取数据文件。这个选项最早出现于MySQL 3.23.25版本。它只用在调试工作中。
- --tmpdir=*dir\_name* 或 -t *dir\_name* 用来存放临时文件的目录的路径名。这个选项的默认值是环境变量TMPDIR的取值；如果你没有设定这个环境变量，则以/tmp为默认值。从MySQL 4.1版本开始，这个选项的值可以是一组将以轮转方式使用的目录——在UNIX系统上，目录名之间要用冒号(:)分隔；在Windows系统上，目录名之间要用分号(;)分隔。
- --update-state 或 -U 对保存在数据表内部的状态标志进行刷新。完好的数据表将被标志为一切正常，有缺陷的数据表将被标记为需要修复。这个选项将使今后的以--check-only-changed选项执行的myisamchk程序在完好的数据表上执行得更有效率。这个选项最早出现于MySQL 3.23.14版本。

### E.3.5 与myisamchk和isamchk有关的变量

下面这些与myisamchk和isamchk程序有关的变量都可以按本附录前面内容里的“设置程序变量”小节所介绍的步骤进行设置。

- `key_buffer_size` 用来存放索引块的缓冲区的长度。
- `read_buffer_size` 读操作的缓冲区的长度。
- `write_buffer_size` 写操作的缓冲区的长度。
- `sort_buffer_size` 用来完成索引键值排序操作的缓冲区的长度。（`--recover`操作会用到这个缓冲区，但`--safe-recover`操作不会用到它。）
- `sort_key_blocks` 这个变量与数据表的索引所使用的二元树数据结构的深度有关。这个值很少需要修改。
- `decode_bits` 在对压缩数据表进行解码时使用的二进制位的个数。这个值越大，解码操作的速度也就越快，但会消耗较多的内存。一般说来，这个变量的默认值已经足以应付大多数情况了。

下面这些变量只适用于myisamchk程序，它们是从MySQL 4.0.0版本开始新增加的：

- `ft_max_word_len` 允许包括在FULLTEXT索引里的单词的最大长度，长于这个长度的单词将被忽略。
- `ft_max_word_len_for_sort` FULLTEXT索引中适合快速排序的单词的最大长度，只有长度小于这个数字的单词才会被ALTER TABLE、CREATE INDEX、REPAIR TABLE等语句用来建立FULLTEXT索引的快速索引创建方法认为是足够短的。长度大于这个数字的单词将使用另一种较慢的方法来插入。
- `ft_min_word_len` 允许包括在FULLTEXT索引里的单词的最小长度，短于这个长度的单词将被忽略。
- `myiasm_block_size` 索引块的区块长度。

### E.4 myisampack和pack\_isam

myisampack和pack\_isam工具程序将生成压缩的只读数据表。在确保你仍能快速访问数据记录的同时，在一般情况下，它们能减少大约40%到70%的存储空间占用量。myisampack程序负责压缩MyISAM数据表，能够对所有的数据列类型进行压缩。pack\_isam程序负责压缩ISAM数据表，只能对不包含BLOB和TEXT数据列的数据表进行压缩。

在MySQL 3.23.19（这是MySQL软件在GNU General Public License（GNU通用公共许可证）下发行的第一个版本）及以后的版本里，myisampack和pack\_isam工具程序可以自由使用；在此之前，只有MySQL软件的用户才能使用它们。因此，你最好选用3.23.19或更高版本的MySQL软件。

MySQL软件的任何版本都能把用这两个工具程序生成的压缩数据表的内容读出来。因此，如果你的应用软件所涉及的数据表只包含只读信息而不需要修改——比如档案或者百科全书之类的东西，就很值得在发行应用软件之前先用这两个工具程序来压缩一下有关的数据表。比如

说，如果你的应用软件使用了嵌入式MySQL服务器且将以光盘的形式发行，对MyISAM数据表进行压缩将使你能够把更多的数据保存在光盘上。

如果想把压缩文件转换为允许修改的非压缩格式，可以使用`myisampack --unpack`（对应于MyISAM数据表）或`pack_isam --unpack`（对应于ISAM数据表）命令。

`myisampack`和`pack_isam`程序只对数据文件进行压缩，对索引文件没有影响。因此，在运行完`myisampack`或`pack_isam`程序后，还必须用`myisampack --recover --quick`或`pack_isam --recover --quick`命令来刷新一下索引。

### E.4.1 用法

```
myisampack [ options ] tbl_name . . .
pack_isam [ options ] tbl_name . . .
```

`tbl_name`参数可以是一个数据表的名字，也可以是数据表的索引文件的名字。（MyISAM数据表的索引文件以`.MYI`为扩展名，ISAM数据表的索引文件以`.ISM`为扩展名。）如果你想对之进行压缩的数据表不在当前目录里，就必须把它们所在的目录的路径名也包括在`tbl_name`参数里。

### E.4.2 myisampack和pack\_isam都支持的标准选项

```
--character-sets-dir  --help  --verbose
--debug               --silent --version
```

`--character-sets-dir`选项只能由`myisampack`程序使用，并且只能用在MySQL 3.23.33及以后的版本里。

### E.4.3 myisampack和pack\_isam都支持的其他选项

- `--backup` 或 `-b` 在对每一个`tbl_name`参数所给定的数据文件进行压缩之前，先制作一个备份。备份文件的名字将是`tbl_name.OLD`。
- `--force` 或 `-f` 对数据表强制进行压缩，不管压缩结果文件的长度是否大于原始文件的长度，也不管是否已经存在有该数据表的一个临时文件。一般说来，如果`myisampack`或`pack_isam`发现已经存在着一个名为`tbl_name.TMD`的文件，它就会在显示一条出错信息后简单地退出执行，因为这通常意味着有另外一个`myisampack`或`pack_isam`程序实例正在运行。但也存在着这样一种可能性：你在这两个工具程序尚未执行完毕的时候强行终止了它们的运行——因为来不及把临时文件安全地删除掉，所以它们就遗留在了系统里。如果你知道你的系统曾发生过这样的事情，就应该用`--force`选项来执行这两个工具程序，让它们不管是否存在临时文件都强行运行。（另一种做法是以手动方式去删除临时文件。）
- `--join=join_tbl` 或 `-j join_tbl` 把你在命令行上给出的所有数据表合并为一个名为`join_tbl`的压缩数据表。参加这种合并操作的数据表必须具有同样的结构（即数据列的名字、类型、索引都必须一模一样）。

- `--test` 或 `-t` 在测试模式里运行。程序将“假装”进行压缩，并把你在真正进行压缩时会看到的各种信息显示出来。
- `--tmpdir=dir_name` 或 `-T dir_name` 用来存放临时文件的目录的路径名。
- `--wait` 或 `-w` (布尔) 如果数据表正被其他客户(程序)使用着，则等待并重试。(如果某个数据表在你对它进行压缩的过程中可能会被修改，就不应该对它进行压缩。)

#### E.4.4 pack\_isam独有的选项

myisampack程序没有独有的选项，pack\_isam程序有一个独有的选项。

- `--packlength=n` 或 `-p n` 用 $n$ 个字节来记录被压缩数据表中的数据记录的长度， $n$ 是一个从1~3的整数。在默认的情况下，pack\_isam程序将自动选定这个长度。但在某些场合，小一些的长度也可以满足需要；此时，pack\_isam程序将显示一条消息来告诉你这一情况。可以再次运行程序并通过`--packlength`选项明确地设定一个较短的长度，这将进一步节省一些空间。

### E.5 mysql

mysql是一个交互式的MySQL客户程序，可以用它来连接MySQL服务器、发出查询命令、查看查询结果。mysql还可以用来以批处理方式执行保存在文件里的查询命令，只需像下面这样把它的输入重定向为某个特定的文件即可：

```
% mysql -u sampadm -p -h cobra.snake.net sampdb < my_query_file
```

在交互模式下，mysql程序会在启动后显示一个“mysql>”提示符以表明它正在等待输入。发出查询命令的办法是这样的：在“mysql>”提示符处敲入有关命令的文本（如有必要，输入的内容允许跨越多行），再敲入一个分号(;)或“\g”作为查询命令的结束标志。mysql程序将把查询命令发送到MySQL服务器去执行，显示查询结果，然后再次显示“mysql>”提示符以表明它在等待你输入下一条命令。“\G”也可以用做查询命令的结束标志，但它将把查询结果按纵向方式显示（即每个数据列值占据一个输出行）。

随着你敲入输入内容，mysql程序会改变提示符以表明它正在等待什么东西。“mysql>”提示符是mysql程序的主提示符，你将在这个提示符下开始输入每一个查询命令。其他提示符（如下表所示）都是辅助提示符，它们表示你还得再输入一些内容才能完成一条完整的查询命令。

提 示 符	含 义
mysql>	等待新查询的第一行
->	等待当前查询的下一行
'>	等待当前查询中单引号字符串的完成
">	等待当前查询中双引号字符串的完成

“'>”和“">”提示符表明你在前面敲入了一个单引号或双引号（以输入一个字符串），但尚未输入与之配对的结束引号。这种情况多发生在你忘记用适当的引号结束一个字符串的场合。

要想退出这个字符串收集模式，请先根据提示符“>”或“>”的提示敲入一个单引号或双引号，再敲入“\c”以取消当前查询命令。

当mysql程序运行在交互模式时，它会把你敲入的查询命令保存到一个历史文件里去。这个文件的默认路径名是\$HOME/.mysql\_history，可以通过环境变量MYSQL\_HISTORY来改变它。你可以把以前输入过的查询命令从命令历史里调出来并再次执行之，在执行之前还可以先对它进行一些编辑。下面是mysql程序比较常用的一些输入行编辑命令。

按键序列/组合键	含 义
上箭头键 或 Ctrl-P	调出前一个输入行
下箭头键 或 Ctrl-N	调出后一个输入行
左箭头键 或 Ctrl-B	向左移动光标
右箭头键 或 Ctrl-F	向右移动光标
Escape Ctrl-B	把光标向左移动一个单词
Escape Ctrl-F	把光标向右移动一个单词
Ctrl-A	把光标移动到输入行的开头
Ctrl-E	把光标移动到输入行的末尾
Ctrl-D	删除光标位置上的字符
Delete	删除光标前面（左侧）的字符
Escape D	删除单词
Escape Backspace	删除光标前面（左侧）的单词
Ctrl-K	从光标位置删除到输入行的末尾
Ctrl-_	取消前一次修改；可多次重复

mysql程序的部分选项——主要是那些将导致mysql程序以非交互方式运行的选项（如--batch、--html、--quick等）——会禁用这种命令历史机制。

在Windows系统上，mysql程序的命令行编辑功能可能无法使用。此时，可以试试mysqlc客户程序，它类似于mysql程序，但它的命令行编辑功能经过特殊的编译处理。（这是个好消息，但还有个坏消息：mysqlc已经很长时间没有升级了，所以它不能理解那些比较新的选项或内部命令。）

mysqlc程序必须有cygwinb19.dll库的支持才能工作。如果你还没有把这个函数库安装到mysqlc程序所在的目录（比如MySQL发行版本的bin目录）里，请从发行版本的lib目录里找到这个文件并把它复制到MySQL软件的bin目录或者你的Windows系统目录里去。

### E.5.1 用法

```
mysql [ options ] [ db_name ]
```

如果你给出了db\_name参数，该数据库就将成为本次会话期间的默认数据库。如果你没有给出db\_name参数，mysql程序将以没有默认数据库的方式启动；而你则必须在今后的查询命令里以db\_name.tbl\_name的形式来指称数据表，或者在发出查询命令之前先用一条USE db\_name命令来指定一个默认数据库。



### E.5.2 mysql支持的标准选项

<code>--character-sets-dir</code>	<code>--host</code>	<code>--silent</code>
<code>--compress</code>	<code>--password</code>	<code>--socket</code>
<code>--debug</code>	<code>--pipe</code>	<code>--user</code>
<code>--default-character-set</code>	<code>--port</code>	<code>--verbose</code>
<code>--help</code>	<code>--set-variable</code>	<code>--version</code>

`--character-sets-dir`和`--default-character-set`选项从MySQL 3.2.15版本开始就可以使用了。从MySQL 4开始,mysql程序还支持各种标准的SSL选项。

在同一条命令里给出多个`--silent`或`--verbose`选项将加强它们的效果。

### E.5.3 mysql独有的选项

- `--auto-rehash` (布尔) 在启动的时候,mysql程序会对数据库、数据表、数据列的名字进行散列计算并构造出一个用来实现名字自动补足功能的数据结构。此后,当输入查询命令的时候,只需输入某个名字的前几个字符再按下键盘上的Tab键,mysql程序就会(如果不会产生二义性的话)自动补足这个名字。在默认情况下,这种对名字进行散列计算的机制是处于激活状态的;`--skip-auto-hash`选项将禁用这一机制,使mysql程序能够更快地启动——尤其是在你有很多数据表的场合。

如果在启动mysql程序时禁用了名字进行散列计算的机制,可稍后又想使用名字自动补足功能,可以在mysql>提示符处使用`rehash`命令。

这个选项最早出现于MySQL 4.0.2版本。在4.0.2版本之前,需要使用`--no-auto-rehash`选项来禁用mysql程序对名字进行散列计算的机制。

- `--batch` 或 `-B` 以批处理方式运行mysql程序。查询结果将显示为制表符间隔格式(每个数据行占据一个输出行,各数据列值之间用制表符分隔)。这种格式的输出生报告非常适合被导入到其他程序(比如电子表格软件)里去做进一步的处理。在默认情况下,查询结果输出报告的第一行将是一个由各数据列名称组成的标题行。如果不想看到这行标题,就需要使用`--skip-column-names`选项。
- `--column-names` (布尔) 在查询结果的输出报告里显示数据列的名字。这个选项最早出现于MySQL 4.0.2版本。
- `--database=db_name` 或 `-D db_name` 设定默认数据库。这个选项最早出现于MySQL 3.22.29版本。
- `--debug-info` 或 `-T` (布尔) 在MySQL程序结束运行时显示调试信息。
- `--execute=query` 或 `-e query` 执行查询命令并退出。必须把查询命令放在引号里以避免shell把查询命令解释为多个命令行参数。mysql程序允许一次给出多个查询命令,但必须在`query`字符串里用分号把它们彼此分隔开。
- `--force` 或 `-f` (布尔) 在从一个文件读取查询命令(即以批处理方式执行)的时候,如果发生错误,mysql程序通常会退出执行。如果给出了这个选项,mysql将忽略这种错误并继续处理查询命令。

- `--html` 或 `-H` (布尔) 生成HTML输出。这个选项可以用在MySQL 3.22.26及以后的版本里。
- `--i-am-a-dummy` (布尔) 这个选项是`--safe-updates`的同义词。它最早出现于MySQL 3.23.11版本。
- `--ignore-spaces` 或 `-i` 让MySQL服务器忽略函数名与引导其输入参数表的左括号字符“(”之间的空格。在默认情况下,左括号字符必须紧跟在函数名的后面,它们之间不允许有间隔。这个选项将使函数名被当做保留字。这个选项最早出现于MySQL 3.22.21版本。
- `--line-numbers` (布尔) 在出错信息里显示行号。这是mysql程序的默认行为;如果不想看到行号,就需要使用`--skip-line-numbers`选项。这个选项最早出现于MySQL 4.0.2版本。在3.22.5到4.0.2版本里,`--skip-line-numbers`或`-L`选项都可以实现不在出错信息里显示行号的目的。
- `--local-infile` (布尔) 允许或者禁用LOAD DATA LOCAL语句。在MySQL 3.23.49及以后的版本里,LOCAL功能(虽然已经出现)的默认设置是处于禁用状态。如果你发出的LOAD DATA LOCAL语句导致了一条出错信息,请在以`--local-infile`选项重新启动mysql程序后再试一次。这个选项还可以用来关闭LOCAL功能(如果它正处于激活状态的话):在MySQL 4.0.2及以后的版本里,使用`--disable-local-infile`选项;在此之前,请使用`--local-infile=0`。

如果MySQL服务器被配置成不允许使用LOCAL功能的情况,这个选项将不会有任何效果。

- `--named-commands` 或 `-G` (布尔) 允许mysql程序的内部命令的长格式形式出现在任何输入行的开头部分。如果你用`--disable-named-commands`选项禁用了这一功能,长格式命令就只允许出现在主提示符处而不允许出现在辅助提示符处——也就是说,它们不允许出现在多行语句的第二及后续各行上。  
这一功能的历史演变是比较复杂的。在MySQL 3.23.22之前的版本里,这一功能是默认激活的,但此后则是默认禁用的。在MySQL 3.23.11到3.23.22版本里,可以用`--no-named-commands`选项来禁用此功能;在3.23.23到4.0.1版本里,可以用`--enable-named-commands`选项来激活之;但从4.0.1版本开始,这一功能的激活或者禁用则是通过布尔选项`--named-commands`来控制的。
- `--no-auto-rehash` 或 `-A` 请参见对`-auto-rehash`选项的介绍;`--no-auto-rehash`从MySQL 4.0.2版本开始已经逐渐被淘汰。
- `--no-beep` 或 `-b` (布尔) 在发生错误时不发出蜂鸣报警音。这个选项最早出现于MySQL 4.0.2版本。
- `--no-named-commands` 或 `-g` 请参见对`--named-commands`选项的介绍;`--no-named-commands`从MySQL 4.0.2版本开始已经逐渐被淘汰。
- `--no-pager` 请参见对`--pager`选项的介绍。这个选项最早出现于MySQL 3.23.28版本,但从MySQL 4.0.2版本开始已逐渐被`--disable-pager`所取代。
- `--no-tee` 请参见对`--tee`选项的介绍。这个选项最早出现于MySQL 3.23.28版本,但从

MySQL 4.0.2版本开始已逐渐被--disable-tee所取代。

- **--one-database 或 -o** 这个选项用在需要根据某个变更日志文件的内容而对数据库做出修改的场合。它告诉mysql程序只对默认数据库（即在命令行上指定的数据库）做出修改，对其他数据库的修改将被忽略。如果没有在命令行上指定数据库，则不进行任何修改。
- **--pager[=*program*]** 使用一个分页显示程序（比如/bin/more或/bin/less）来分页显示比较长的查询结果，每次显示一页。如果没有给出*program*参数，mysql程序将根据环境变量PAGER的值来决定将使用哪一个分页显示程序。查询结果的分页显示功能在批处理模式下不可用，在Windows系统上也不可用。可以用--disable-pager选项来禁用这项功能。  
--pager选项最早出现于MySQL 3.23.28版本。在4.0.2之前的版本里，这项功能需要使用--no-pager而不是--disable-pager来禁用。
- **--prompt=*str*** 把mysql程序的主提示符从“mysql>”改变为由*str*定义的字符串。这个字符串可以包含一些特殊的序列，具体情况见第E.5.6节。这个选项最早出现于MySQL 4.0.2版本。
- **--quick 或 -q** 在默认情况下，mysql程序将在从MySQL服务器把查询命令的结果集全部检索出来之后才开始显示它们。这个选项将使mysql程序每检索出一个数据行就立刻显示之，这既减少了内存的占用量，又能使那些不这样做就会失败的大查询得以成功地完成。不过，最好不要在交互模式下使用这个选项；因为如果用户暂停了输出或者挂起了mysql程序的话，MySQL服务器就会进入等待状态，进而影响到其他客户（程序）的操作。
- **--raw 或 -r（布尔）** 把数据列值原封不动地显示出来，不对其中的特殊字符做任何转义处理。这个选项通常要与--batch选项联合使用。
- **--safe-updates 或 -U（布尔）** 这个选项给数据库的修改操作加上了一些限制，这些限制对MySQL新手很有好处。如果你激活了这个选项，MySQL将只允许会改变数据库数据的语句（比如UPDATE和DELETE语句）在以下两种情况下执行：1）将被修改的记录是通过索引键值而确定的；2）使用了LIMIT子句。这有助于防止查询命令意外地改变或者删除数据表的全部或者大部分内容。此外，这个选项还将把非关联检索（即对单个数据表进行的检索）或关联检索（即对多个数据表进行的检索）的结果集分别限制在1千或1百万个数据行以下；这两个上限值可以通过变量select\_limit和max\_join\_size加以改变。这个选项最早出现于MySQL 3.23.11版本。
- **--skip-column-names 或 -N** 在查询结果的输出报告里不显示以数据列名字充当的输出列标题。重复两次给出--silent选项也能得到同样的效果。这个选项最早出现于MySQL 3.22.20版本，但它的-N形式从MySQL 4.0.2版本开始已经逐渐被淘汰。另请参见对--column-names选项的介绍。
- **--skip-line-numbers 或 -L** 不在出错信息里显示行号。这个选项最早出现于MySQL 3.22.5版本，但它的-L形式从MySQL 4.0.2版本开始已经逐渐被淘汰。另请参见对--line-names选项的介绍。
- **--table 或 -t（布尔）** 把输出报告显示为表格形式，即各行之间用表格线分隔且纵向对齐。这是运行在非批处理模式中的mysql程序的默认输出格式。

- `--tee=file_name` 把全部输出信息的一份副本追加到指定文件的末尾。可以用`--disable-tee`选项来禁用这项功能。这个选项不能工作在批处理模式下。它最早出现于MySQL 3.23.28版本。在4.0.2之前的版本里，这项功能需要使用`--no-tee`而不是`--disable-tee`来禁用。
- `--unbuffered` 或 `-n` (布尔) 每执行完一个查询命令，就对mysql程序用来与MySQL服务器进行通信的缓冲区进行“冲洗”（“冲洗”缓冲区的意思是把缓冲区的内容写到磁盘上去或者把有关信息从磁盘上读到缓冲区里）。
- `--vertical` 或 `-E` 按纵向方式显示查询结果，即查询结果中的每一条记录将被显示为这样一组输出行：1）每组输出行的第一行显示着那条记录在结果集里的序号；2）其他输出行依次对应着那条记录的各个字段，由字段名和字段值两部分构成。如果查询命令检索出来的记录很长，把它们显示为纵向格式将增加可读性。  
如果在启动mysql程序时没有给出这个选项，但在后来执行某个查询命令时又想使用纵向显示格式，就需要使用“\G”而不是“;”或“\g”来作为查询命令的结束符。  
这个选项最早出现于MySQL 3.22.5版本。
- `--wait` 或 `-w` 如果mysql程序无法建立与MySQL服务器的连接，则等待并重试。
- `--xml` 或 `-X` (布尔) 生成XML输出。这个选项最早出现于MySQL 4.0.0版本。

#### E.5.4 与mysql有关的变量

下面这些与mysql程序有关的变量都可以按本附录前面内容里的“设置程序变量”小节所介绍的步骤进行设置。

- `connect_timeout` 如果在经过这么多秒之后还没有建立起与MySQL服务器的连接，则放弃这次尝试。这个变量最早出现于MySQL 3.23.28版本。
- `max_allowed_packet` mysql程序与MySQL服务器进行通信时使用的缓冲区的最大长度。
- `max_join_size` 如果mysql程序在启动时使用了`--safe-updates`选项，这个变量的值就是那些对多个数据表进行关联检索的SELECT语句所能返回的数据行的最大个数。这个变量最早出现于MySQL 3.23.11版本。
- `net_buffer_length` mysql程序与MySQL服务器进行通信时使用的缓冲区的初始长度。这个缓冲区可以扩大到`max_allowed_packet`个字节长。
- `select_limit` 如果mysql程序在启动时使用了`--safe-updates`选项，这个变量的值就是那些对单个数据表进行检索的SELECT语句所能返回的数据行的最大个数。这个变量最早出现于MySQL 3.23.11版本。

#### E.5.5 mysql命令

除允许向MySQL服务器发送SQL语句外，mysql程序本身还有一些内部命令。在给出这些命令的时候，必须把有关内容全都写在同一行上。可以在这些命令的末尾加上一个分号作为结束符，但MySQL并不要求你必须这样做。这些命令中的大多数都有一个长格式形式（一个单词）和一个短格式形式（一个反斜线加一个字符）。长格式命令不区分字母的大小写情况，短格式命



令则必须按下面有关内容中的字母大小写形式写出。

注意，如果禁用了“允许mysql程序的内部命令的长格式形式出现在任何输入行的开头部分”功能（例如，通过--disable-named-commands选项），长格式命令就只允许出现在主提示符“mysql>”处而不能出现在辅助提示符处——也就是说，它们不能出现在多行语句的第二及后续各行上。

- clear 或 \c 清除（撤销）当前查询。所谓“当前查询”指的是你正在敲入的查询语句；那些已经被发送给MySQL服务器或者mysql程序已经开始显示其查询结果的查询是无法用这个命令来撤销的。
- connect [ db\_name [ host\_name ] ] 或 \r [ db\_name [ host\_name ] ] 连接（或者再次连接）指定主机上的指定数据库。如果没有给出数据库名或主机名，则使用当前mysql会话中最近一次用过的值。
- edit 或 \e 对当前查询进行编辑。mysql程序将依次检查环境变量EDITOR和VISUAL的值以决定要使用哪一个编辑器。如果这两个环境变量都没有定义，mysql程序将使用vi编辑器。这个选项在Windows系统上不可用。
- ego 或 \G 把当前查询发送到MySQL服务器并按纵向格式显示其查询结果。这个命令最早出现于MySQL 3.22.11版本。
- exit 与quit命令作用相同。
- go 或 \g 把当前查询发送到MySQL服务器并显示其查询结果。
- help 或 \h 或 ? 显示帮助信息，这些信息对mysql程序各种内部命令做了简单的介绍。
- nopager 或 \n 禁用分页显示机制，把输出发送到标准输出设备上去。这个命令最早出现于MySQL 3.23.28版本，但在Windows系统上不可用。
- notee 或 \t 不把输出内容追加到tee文件的末尾。这个命令最早出现于MySQL 3.23.28版本。
- pager [ program ] 或 \P [ program ] 把输出内容重定向到本命令的program参数或者系统的PAGER环境变量所指定的分页显示程序去。这个命令最早出现于MySQL 3.23.28版本，但在Windows系统上不可用。
- print 或 \p 显示当前查询的文本（只显示查询命令本身，不显示执行这条查询命令而获得的结果）。
- prompt arguments 或 \R arguments 重新定义mysql程序的主提示符。新提示符字符串将由从prompt关键字后面第一个空格开始的所有字符（包括其他空格）构成。这个字符串可以包含一些特殊的序列，具体情况见第E.5.6节。如果想把提示符恢复为它的默认值“mysql>”，请发出一条不带任何参数的prompt或\R命令。这个命令最早出现于MySQL 4.0.2版本。
- quit 或 \q 退出mysql程序。
- rehash 或 \# 重新对数据库、数据表、数据列的名字进行散列计算，计算结果将用于实现这些名字的自动补足功能。另请参见对--auto-rehash选项的介绍。
- source file\_name 或 \. file\_name 从指定文件里读取并执行SQL查询命令。注意，Windows



路径名中的反斜线字符 (\) 必须写成斜线字符 (/)。这个命令最早出现于MySQL 3.23.9版本。

- `status` 或 `\s` 检索并显示来自MySQL服务器的状态信息，比如服务器的版本、当前数据库、当前连接是否是安全化连接等等。
- `system command` 或 `\! command` 用默认shell来执行`command`命令。这个命令最早出现于MySQL 4.0.1版本，但在Windows系统上不可用。
- `tee file_name` 或 `\T file_name` 把输出内容追加到指定文件的末尾。这个命令最早出现于MySQL 3.23.28版本。
- `use db_name` 或 `\u db_name` 把指定数据库选定为当前的默认数据库。

### E.5.6 mysql提示符定义序列

在默认情况下，mysql程序的主提示符是“mysql>”，但可以通过环境变量MYSQL\_PS1、mysql程序的--prompt选项或者它的prompt内部命令来重新定义之。下面是一些允许用在提示符定义里的、具有特殊含义的转义序列：

转义序列	含 义
<code>\c</code>	当前输入行
<code>\d</code>	当前数据库的名字；如果尚未选定数据库，则是“(none)”
<code>\D</code>	完整的日期和时间
<code>\h</code>	当前主机
<code>\m</code>	分钟
<code>\o</code>	月份数字
<code>\O</code>	月份名称，三个字母
<code>\P</code>	当前端口号、套接字名或者命名管道的名字
<code>\p</code>	a.m. / p.m. 标志
<code>\r</code>	小时（12小时制）
<code>\R</code>	小时（24小时制）
<code>\s</code>	秒
<code>\S</code>	分号
<code>\t</code>	制表符
<code>\u</code>	当前用户名；不带主机名
<code>\U</code>	当前用户名；带主机名
<code>\v</code>	MySQL服务器的版本号
<code>\y</code>	年（2位数字）
<code>\Y</code>	年（4位数字）
<code>\w</code>	星期几；三个字母
<code>\'</code>	单引号
<code>\"</code>	双引号
<code>\_</code>	空格字符
<code>\ </code>	空格字符（这个转义序列是一个反斜线加一个空格）
<code>\\</code>	反斜线字符 “\”
<code>\n</code>	换行符
<code>\x</code>	字符x，x是上面没有列出的任何字符

## E.6 mysqlaccess

这个脚本允许你去连接MySQL服务器、检索访问权限信息、测试向用户授权的结果。这些工作都是在来自mysql数据库的user、db和host数据表的副本上进行的。（但mysqlaccess脚本不能对数据表级或数据列级的权限进行测试。）如果对测试结果感到满意，还可以把在这些临时数据表上做出的修改写入mysql数据库里的各有关数据表去。

要想使用mysqlaccess脚本，必须拥有访问各有关权限数据表所必须的权限才行。

### E.6.1 用法

```
mysqlaccess [ host_name [ user_name [ db_name ] ] ] options
```

### E.6.2 mysqlaccess支持的标准选项

```
--host          --password      --user          --version
```

### E.6.3 mysqlaccess独有的选项

- --brief 或 -b 以单行格式显示结果。
- --commit 把临时权限表的内容复制到mysql数据库里去。记住：要想让MySQL服务器注意到你做出的修改，还得再执行一个mysqladmin flush-privileges命令才行。
- --copy 把权限表的内容加载到临时数据表里去。
- --db=db\_name 或 -d db\_name 数据库的名字。
- --debug=n 设定调试级别。n必须是一个0~3之间的整数；这个数字越大，mysqlaccess脚本给出的诊断性输出信息也就越多。
- --howto 显示一些演示mysqlaccess脚本用法的示例。
- --old\_server 如果你的MySQL服务器版本早于3.21，则需要使用这个选项。它将使mysqlaccess脚本对它发送给MySQL服务器的查询命令做出一些必要的调整。
- --plan 显示一份功能清单，这份清单里的功能将在今后的mysqlaccess脚本里得到实现。
- --preview 对照显示真正的权限表和临时数据表之间的差异。
- --relnotes 显示mysqlaccess脚本的发行信息。
- --rhost=host\_name 或 -H host\_name mysqlaccess脚本将与之进行连接的远程服务器主机的名字。
- --rollback 复原（撤销）对临时权限表所做的修改。
- --spassword=pass\_val 或 -P pass\_val MySQL超级用户（即有权修改权限表的用户）的口令。
- --superuser=user\_name 或 -U user\_name MySQL超级用户的用户名。
- --table 或 -t 以表格形式显示结果。

## E.7 mysqladmin

mysqladmin程序能够让你在客户端通过与MySQL服务器的通信去完成各种数据库管理工作。可以利用mysqladmin程序去获取服务器操作信息、控制服务器操作、设置口令、创建或丢弃数据库。

### E.7.1 用法

```
mysqladmin [ options ] command . . .
```

### E.7.2 mysqladmin支持的标准选项

--character-sets-dir	--password	--socket
--compress	--pipe	--user
--debug	--port	--verbose
--help	--set-variable	--version
--host	--silent	

如果使用了--silent选项，mysqladmin程序在无法与MySQL服务器建立连接的时候就会默默地退出执行。--verbose选项是从MySQL 3.22.30版本开始新增的，它将使mysqladmin程序在执行某些内部命令时显示更多的信息。--character-sets-dir选项是从MySQL 3.23.21版本开始新增的。从MySQL 4开始，mysqladmin程序还支持各种标准的SSL选项。

### E.7.3 mysqladmin独有的选项

- --count=*n* 或 -c *n* 与--sleep选项一起使用，负责给出mysqladmin程序将要循环执行的次数。这个选项最早出现于MySQL 4.0.3版本。
- --force 或 -f (布尔) 这个选项有两个效果：其一，让mysqladmin程序在执行drop *db\_name*命令时不进行确认；其二，当你在命令行上同时给出了多个命令时，mysqladmin程序将尽量执行每一个命令而不管它们当中是否有执行出错的。（在正常情况下，mysqladmin程序将在第一次出错后退出执行。）
- --relative 或 -r (布尔) 与--sleep选项一起使用，对照显示mysqladmin程序前、后两次执行结果的不同之处。这个选项目前只与extend-status命令一起使用才有效果。
- --sleep=*n* 或 -i *n* 每隔*n*秒重复执行一次你在mysqladmin命令行上给出的命令，重复次数由--count选项决定。
- --timeout=*n* 或 -t *n* 如果在经过了*n*秒之后还没能连接上MySQL服务器，则放弃本次连接尝试。这个选项最早出现于MySQL 3.22.1版本，但在3.23.29及以后的版本里因为新增了connect\_timeout变量的缘故又被去掉了。
- --vertical 或 -E (布尔) 这个选项与--relative选项的功能相同，但将把输出内容按纵向格式显示。它最早出现于MySQL 3.23.14版本。
- --wait[=*n*] 或 -w[*n*] 如果无法建立与MySQL服务器的连接，则等待*n*秒之后再进行重试。

如果没有进行设定,  $n$  的默认值将是1。如果使用 `-w` 来设置  $n$  值, 它们之间不允许有空格, 否则  $n$  值将无法得到正确的解释。

#### E.7.4 与mysqladmin有关的变量

下面这些与mysqladmin程序有关的变量都可以按本附录前面内容里的“设置程序变量”小节所介绍的步骤进行设置。

- `connect_timeout` 如果在经过了`connect_timeout`秒之后还没能连接上MySQL服务器, 则放弃本次连接尝试。这个变量最早出现于MySQL 3.23.29版本; 因为它的出现, `--timeout`选项被去掉了。
- `shutdown_timeout` 要求shutdown命令必须在`shutdown_timeout`秒内成功地关闭MySQL服务器, 否则报告出错。这个变量最早出现于MySQL 3.23.34版本。

#### E.7.5 mysqladmin命令

在mysqladmin命令行上, 还可以在选项(如果有的话)的后面给出一个或者多个下列命令。在不引起二义的前提下, MySQL允许只写出命令的前几个字符。比如说, `processlist`命令允许简写为“`process`”或“`proc`”, 但不允许简写为“`p`”。

部分mysqladmin命令有着与之功能相同的SQL语句, 具体情况见各有关条目中的说明。对那些SQL语句的详细说明请参见本书的附录D。

- `create db_name` 以给定名字创建一个新数据库。这个命令与`CREATE DATABASE db_name`语句等价。
- `drop db_name` 删除指定的数据库以及数据库里的数据表。在打算使用这个命令的时候, 请一定要三思而后行, 因为你无法把删除的数据库再找回来。如果没有使用`--force`选项, mysqladmin程序将要求你确认这个命令。这个命令与`DROP DATABASE db_name`语句等价。
- `debug` 让MySQL服务器显示调试信息。
- `extended-status` 依次显示MySQL服务器各状态变量的名字和值。这个命令与`SHOW STATUS`语句等价。它最早出现于MySQL 3.22.10版本。
- `flush-hosts` 清空主机缓存区。这个命令与`FLUSH HOSTS`语句等价。
- `flush-logs` 清空(关闭再打开)日志文件。这个命令与`FLUSH LOGS`语句等价。
- `flush-privileges` 重新加载各个权限表。这个命令与`FLUSH PRIVILEGES`语句等价。它最早出现于MySQL 3.22.12版本。
- `flush-status` 对状态变量进行清零(它将把多个计数器重置为0)。这个命令与`FLUSH STATUS`语句等价。
- `flush-tables` 清空数据表缓存区。这个命令与`FLUSH TABLES`语句等价。
- `flush-threads` 清空线程缓存区。
- `kill id,id,...` 杀掉(终止执行)指定的服务器线程。如果同时给出了多个线程ID号, 它

们之间不允许有任何空格以免它们被MySQL误认为是跟在kill命令后的其他命令。可以用mysqladmin processlist命令查出当前都有哪些线程正在运行。这个命令与使用KILL语句去逐个地杀掉各个线程的做法等价。

- password *new\_password* 改变运行mysqladmin程序时使用的账户的口令。(这里隐含着一个假设:既然你能够用这个账户连接上MySQL服务器,就证明你知道这个账户的口令。)新口令将被设置为*new\_password*。这个命令与SET PASSWORD语句等价。
- ping 检查MySQL服务器是否正在运行。
- processlist 列出一份当前正在执行的服务器进程的名单。这个命令与SHOW PROCESSLIST语句等价。如果你还使用了--verbose选项,那么这个命令将与SHOW FULL PROCESSLIST语句等价。
- refresh 这个命令将清空数据表缓存区和各个权限表,同时还会先关闭再打开各有关日志文件。如果这个服务器是镜像机制中的主服务器,这个命令将使它删除在二进制日志索引文件里列出的各个二进制变更日志并把二进制日志索引文件的长度截短为0。如果这个服务器是镜像机制中的从服务器,这个命令将使它忘记自己在主日志里的位置。
- reload 重新加载各个权限表。这个命令与FLUSH PRIVILEGES语句等价。
- shutdown 关闭MySQL服务器。
- start-slave 启动一个镜像从服务器。这个命令与SALVE START语句等价,它最早出现于MySQL 3.23.16版本。
- status 按短格式显示MySQL服务器的状态信息。
- stop-slave 关闭一个镜像从服务器。这个命令与SALVE STOP语句等价,它最早出现于MySQL 3.23.16版本。
- variables 依次显示MySQL服务器各变量的名字和值。这个命令与SHOW VARIABLES语句等价。从开始区别对待全局级变量和会话级变量的MySQL 4.0.3版本开始,这个命令与SHOW GLOBAL VARIABLES语句等价。(没有与SHOW SESSION VARIABLES语句等价的mysqladmin命令,因为这毫无意义。)
- version 检索并显示MySQL服务器的版本信息字符串,这个字符串与VERSION()函数的返回值完全相同(请参见附录C)。

## E.8 mysqlbinlog

mysqlbinlog程序将以人们能够阅读和理解的格式把二进制变更日志文件的内容显示出来。这个程序既可以直接读出本地主机上的日志文件,也可以连接到某个远程MySQL服务器并读出那台服务器主机上的日志。

二进制变更日志的格式一直处于变化和改进当中。为避免兼容性问题,你使用的mysqlbinlog程序的版本至少应该不低于你的MySQL服务器版本。

### E.8.1 用法

```
mysqlbinlog [ options ] file_name . . .
```



## E.8.2 mysqlbinlog支持的标准选项

```
--debug      --host      --port      --version
--help       --password  --user
```

--host、--password、--port和--user选项用在你想连接某个远程服务器以访问其二进制日志的场合。如果你没有给出这几个选项，mysqlbinlog程序将直接读取本地主机上的日志文件；在这一过程中，它不需要连接任何MySQL服务器。

## E.8.3 mysqlbinlog独有的选项

- --database=db\_name 或 -d db\_name 从日志文件里只提取与指定数据库有关的语句。这个选项只在mysqlbinlog程序读取本地日志时有效。
- --offset=n 或 -o n 跳过日志文件中的前n项记录。
- --position=n 或 -j n 从位置n开始读取日志文件。
- --result-file=file\_name 或 -r file\_name 把输出内容写到指定的文件里去。
- --short-form 或 -s 只显示日志中的语句；日志中与该语句有关的其他信息将不显示。
- --table=tbl\_name 或 -t tbl\_name 获得指定数据表的一份“纯”导出版本，不进行任何转义或修饰性处理。

## E.9 mysqlbug

*MySQL Reference Manual* (MySQL参考手册)对程序漏洞报告的填写流程做了详细的介绍，按照该流程来填写报告能保证它包含足够的信息，便于有关人员解决你发现的那个漏洞。而这个流程中的关键步骤之一就是使用mysqlbug脚本。当你发现与MySQL有关的程序漏洞的时候，只需运行mysqlbug脚本，它就能创建一个标准的漏洞报告并把它发送到MySQL邮件列表去。mysqlbug脚本先把你的系统和MySQL的配置信息收集起来，然后放到一个编辑器里，这个编辑器里的内容就是将要发送出去的邮件消息。(mysqlbug脚本将依次检查VISUAL和EDITOR环境变量的值以确定将要使用哪一个编辑器。如果这两个变量都没有定义，mysqlbug脚本将使用emacs编辑器。)

对编辑器里的邮件消息进行编辑，把你认为有用的信息尽可能多地添加进去，然后存盘并退出编辑器。mysqlbug脚本将问你是否想立刻发送这份报告，如果你回答是，就会马上把它发送出去。

请使用mysqlbug脚本来报告程序漏洞，但千万不要草率地使用它。在很多场合，你发现的“漏洞”要么根本就不是一个真正的漏洞，要么早已在*MySQL Reference Manual* (MySQL参考手册)里有了解决办法。MySQL邮件列表上的资料档案也是一个非常有用的信息源，它能帮你判断你观察到的程序行为到底是不是错误的。可以在网址<http://www.mysql.com/documentation/>上找到对应于MySQL参考手册和MySQL邮件列表资料档案的链接。

mysqlbug是一个shell脚本，所以不能用在Windows系统上。如果你使用的是WinMySQL Admin或MySQLCC等Windows软件，可以使用它们的出错报告功能。此外，在MySQL发行版

本的顶级目录里还有一个名为mysqlbug.txt的文件，可以把它当做一个模板来使用：填上有关信息并把它发送到地址bugs@lists.mysql.com去即可。

## 用法

```
mysqlbug [ address ]
```

程序漏洞报告将默认地被发送到MySQL邮件列表去。但如果你在命令行上给出了一个电子邮件地址，程序漏洞报告就将被发送到指定的地址去。如果你给出的是你本人的电子邮件地址，程序漏洞报告就将被发送到你本人的邮箱而不是MySQL邮件列表；当你第一次使用mysqlbug脚本、还不太了解它的工作情况时，这不失为慎重之举。

## E.10 mysqlcheck

mysqlcheck是一个用来检查和修复数据表的工具程序。它为CHECK TABLE、ANALYZE TABLE、OPTIMIZE TABLE、REPAIR TABLE等语句提供了一个命令行接口。它与myisamchk程序有几分相似，但它是在MySQL服务器仍在运行时使用的。mysqlcheck程序通过发送管理性查询命令到MySQL服务器去执行的办法来完成工作；这与myisamchk程序形成了鲜明的对照——myisamchk直接在数据表文件上进行操作，所以必须由你去负责协调MySQL服务器对各有关数据表的访问或者干脆暂时停止MySQL服务器的运行。

mysqlcheck程序最早出现于MySQL 3.23.38版本。mysqlcheck程序的各个选项全都可以用在MyISAM数据表上。mysqlcheck程序还可以对BDB数据表进行分析；从MySQL 3.23.40版本开始，它还可以对InnoDB数据表进行分析。

### E.10.1 用法

mysqlcheck程序有三种运行模式：

```
mysqlcheck [ options ] db_name [ tbl_name ] . . .
mysqlcheck [ options ] --databases db_name . . .
mysqlcheck [ options ] --all-databases
```

在第一种模式里，mysqlcheck程序将对指定数据库里的指定数据表进行检查；如果没有给出数据表的名字，mysqlcheck将依次对数据库里的所有数据表进行检查。在第二种模式里，mysqlcheck程序把所有的参数都解释为数据库的名字，它将依次检查各指定数据库里的所有数据表。在第三种模式里，mysqlcheck程序将依次检查所有数据库里的所有数据表。

### E.10.2 mysqlcheck支持的标准选项

--character-sets-dir	--host	--socket
--compress	--password	--user
--debug	--pipe	--verbose
--default-character-set	--port	--version
--help	--silent	

从MySQL 4开始, mysqlcheck程序还支持各种标准的SSL选项。

### E.10.3 mysqlcheck独有的选项

下列选项控制着mysqlcheck程序将如何对数据表进行处理。某些处理工作完全可以通过相应的SQL语句来完成;在介绍完mysqlcheck程序的选项之后,我们还将对这些选项与SQL语句的等价对应关系进行介绍。

- `--all-databases` 或 `-A` (布尔) 依次检查所有数据库里的所有数据表。
- `--analyze` 或 `-a` 发出ANALYZE TABLE语句,对数据表进行分析。(比如说,这个选项将对键值的分布情况进行分析。)分析结果将有助于MySQL服务器更快地完成基于索引的查找和关联查找操作。
- `--all-in-1` 或 `-I` (布尔) 如果没有使用这个选项,mysqlcheck程序将为每一个数据表分别发出一个查询。如果使用了这个选项,mysqlcheck程序将按数据库对数据表进行归组,即用同一个查询对同一个数据库里的所有数据表进行检查。
- `--auto-repair` (布尔) 如果在检查过程中发现某些数据表有问题,mysqlcheck程序将在这次检查完毕后再执行一遍以自动修复它们。
- `--check` 或 `-c` 发出CHECK TABLE语句,检查数据表中是否有错误。如果你没有明确地告诉mysqlcheck说你想让它做些什么,这就将是它的默认行为。
- `--check-only-changed` 或 `-C` 只对在上次检查之后又发生过修改或者没有被正确关闭的数据表进行检查。
- `--databases` 或 `-B` (布尔) 把所有的参数都解释为数据库的名字,依次检查各指定数据库里的所有数据表。
- `--extended` 或 `-e` (布尔) 对数据表进行全面检查。如果与`--repair`选项联合使用,mysqlcheck将使用一种比单独给出`--repair`选项时更全面但也更慢的修复方法。
- `--fast` 或 `-F` (布尔) 只对没有被正确关闭的数据表进行检查。
- `--force` 或 `-f` (布尔) 强行继续执行,不管是否出错。
- `--medium-check` 或 `-m` 对数据表进行中级检查。此时使用的数据表检查方法要比给出`--extended`选项时快一些,但不那么全面。这个检查模式已经足以应付大多数场合。
- `--optimize` 或 `-o` 发出OPTIMIZE TABLE语句,对数据表进行优化。
- `--quick` 或 `-q` (布尔) 对于数据表检查操作,这个选项将省略对数据行中的链接进行检查的步骤。如果是与`--repair`选项联合使用,这个选项将只修复索引文件而不触及数据文件。重复写出两遍这个选项与只写出一遍的效果是一样的;这是与myisamchk程序的又一个不同之处(在使用myisamchk程序的时候,把这个选项写两遍的效果与只写一遍的效果是不一样的)。
- `--repair` 或 `-r` 发出REPAIR TABLE语句,对数据表进行修复。这个修复模式能够纠正绝大多数问题,但对惟一化索引中的重复键值问题无能为力。
- `--tables` 改写`--databases`选项。

- `--use-frm` (布尔) 与`--repair`选项一起使用, 让数据表修复操作使用`.frm`文件来解释数据文件和重建索引文件。这个选项主要用在索引文件已丢失或者被损坏的场合。它最早出现于MySQL 4.0.5版本。

mysqlcheck程序的各有关选项与相应的SQL命令的等价对应关系见以下几个表格。

数据表检查选项 (只适用于MyISAM和InnoDB数据表):

mysqlcheck程序选项	相应的SQL语句
<code>--check</code>	<code>CHECK TABLE tbl_list</code>
<code>--check-only-changed</code>	<code>CHECK TABLE tbl_list CHANGED</code>
<code>--extended</code>	<code>CHECK TABLE tbl_list EXTENDED</code>
<code>--fast</code>	<code>CHECK TABLE tbl_list FAST</code>
<code>--medium-check</code>	<code>CHECK TABLE tbl_list MEDIUM</code>
<code>--quick</code>	<code>CHECK TABLE tbl_list QUICK</code>

数据表分析选项 (只适用于MyISAM和BDB数据表):

mysqlcheck程序选项	相应的SQL语句
<code>--analyze</code>	<code>ANALYZE TABLE tbl_list</code>

数据表修复选项 (只适用于MyISAM数据表):

mysqlcheck程序选项	相应的SQL语句
<code>--repair</code>	<code>REPAIR TABLE tbl_list</code>
<code>--repair --quick</code>	<code>REPAIR TABLE tbl_list QUICK</code>
<code>--repair --extended</code>	<code>REPAIR TABLE tbl_list EXTENDED</code>
<code>--repair --use-frm</code>	<code>REPAIR TABLE tbl_list USE_FRM</code>

数据表优化选项 (只适用于MyISAM数据表):

mysqlcheck程序选项	相应的SQL语句
<code>--optimize</code>	<code>OPTIMIZE TABLE tbl_list</code>

## E.11 mysql\_config

mysql\_config是以C语言开发MySQL应用程序时使用的一个辅助性工具程序。这个程序能让你获得在编译C语言源文件或链接MySQL开发库时必须用到的编译标志或链接标志。

mysql\_config程序最早出现于MySQL 3.23.21版本。

### E.11.1 用法

```
mysql_config [ options ]
```

### E.11.2 mysql\_config独有的选项

- `--cflags` 显示访问MySQL头文件所必需的头文件目录标志。
- `--embedded` 或 `--embedded-libs` 这两个选项是`--libmysqld-libs`选项的同义词。
- `--libs` 显示链接MySQL客户程序开发库所必需的函数库链接标志。
- `--sockets` 显示默认的MySQL套接字文件的路径名。
- `--port` 显示默认的MySQL TCP/IP端口号。
- `--version` 显示MySQL版本字符串。
- `--libmysqld-libs` 显示链接嵌入式MySQL服务器开发库所必需的函数库链接标志。

## E.12 mysqld

`mysqld`就是MySQL服务器程序。它是MySQL客户程序访问MySQL数据库的桥梁，要是MySQL服务器没有运行，客户程序就无法使用该服务器所管理的数据库。在启动的时候，`mysqld`将打开一些端口并开始在其上等待客户来与之连接。`mysqld`是一个多线程的程序，它将使用不同的线程来处理各客户连接，使多个客户程序可以并发地得到处理。对数据库进行写操作的查询都将以原子化方式处理；也就是说，当服务器开始执行一个这样的查询时，任何涉及到它正在处理的数据的查询都将被阻塞，直到当前查询执行完毕为止。比如说，两个客户程序不可能同时修改同一数据表里的同一个数据行。

### E.12.1 用法

`mysqld`程序最常见的启动方式很简单，只需写出服务器程序的名字和你想使用的选项就可以了，如下所示：

```
mysqld [ options ]
```

在基于Windows NT的系统上，还可以把MySQL服务器安装并运行作为一项服务。请看下面两条命令：第一条命令把`mysqld-nt`服务器安装并运行作为一项服务，让它在系统开机启动时自动运行；第二条命令则删除了这项服务：

```
C:\> mysql-nt --install
```

```
C:\> mysql-nt --remove
```

默认的服务名是MySql。从MySQL 4.0.2版本开始，还可以在选项的后面另行指定一个服务名，如下所示：

```
C:\> mysql-nt --install service_name
```

```
C:\> mysql-nt --remove service_name
```

这就使多个MySQL服务器能够以不同的服务名同时运行。如果没有给出`service_name`参数，MySQL服务器将以MySql作为自己的服务名，并会在启动时读取选项组[mysql]里的各个文件中的选项。如果给出了`service_name`参数，MySQL服务器就将以该参数作为自己的服务名，并会在启动时读取选项组[`service_name`]里的各个文件中的选项。



从MySQL 4.0.3版本开始，还可以在服务器程序名的后面用--defaults-file选项再额外指定一个选项文件；这样，除标准的选项文件外，MySQL服务器还将在启动时读取该文件里的选项。如下所示：

```
C:\> mysqld-nt --install service_name --defaults-file=file_name
```

在上面这种场合里，*service\_name*参数不允许省略。

以上对--install选项的讨论也适用于--install-manual选项。（此外，允许在--install选项的后面再写出一些参数的做法现在已经被添加到从MySQL 3.23.54版本开始的3.23系列版本里了；换句话说，可以在MySQL 3.23.54及以后的版本里在--install选项的后面再写出一些参数。）

### E.12.2 mysqld支持的标准选项

--character-sets-dir	--help	--socket
--debug	--port	--user
--default-character-set	--set-variable	--version

--character-sets-dir和--default-character-set最早出现于MySQL 3.23.14版本。从MySQL 4开始，mysqld程序还支持各种标准的SSL选项。

注意，虽然MySQL现在已经支持使用--socket选项，但目前仍不支持使用相应的短格式（-S）。

在UNIX系统上，如果你使用了--user选项，MySQL服务器就将使用该账户的用户名来运行。此时，在启动的时候，MySQL服务器将从口令文件里查出该账户的用户ID和用户组ID，并把它们用做自己的用户ID和用户组ID。如此启动的MySQL服务器在运行时将不再具备root权限——它在运行时的权限将由指定账户来决定。（不过，要想让--user选项起作用，MySQL服务器就必须按root用户来启动，因为只有这样才能改变自己的用户ID和用户组ID。）从MySQL 4.0.2版本开始，--user选项的值允许是一个数值形式的用户ID。

### E.12.3 mysqld独有的选项

下列选项是MySQL服务器的通用选项。在随后的几个小节里，我们将依次介绍MySQL服务器的几组专用选项，即它在Windows系统上的独有选项、对应于特定数据表处理程序的独有选项以及与镜像机制有关的选项。

- --ansi 或 -a 这个选项将使MySQL服务器在遇到某些特定类型的SQL语句时按ANSI标准而不是按MySQL标准来采取行动。这个选项将使MySQL服务器的行为与ANSI标准兼容。这个选项最早出现于MySQL 3.23.6版本。它相当于在使用--sql-mode选项的同时还给出了REAL\_AS\_FLOAT、PIPES\_AS\_CONCAT、ANSI\_QUOTES、IGNORE\_SPACE、SERIALIZE和ONLY\_FULL\_GROUP\_BY标志。
- --basedir=*dir\_name* 或 -b *dir\_name* MySQL安装目录的路径名。以相对路径给出的、与MySQL服务器有关的其他路径有很多是以这个目录作为顶级目录的。
- --big-tables 使MySQL服务器能够处理大结果集，它将把临时结果全都保存到磁盘上而不是放在内存里。如果没有使用这个选项，那么当没有足够的内存来容纳大结果集时，就经常会发生“table full”（数据表满）错误。但从MySQL 3.23版本开始，这个选项已经

不再是必需的了。

- `--bind-address=ip_addr` 绑定到给定的IP地址。一般情况下, `mysqld`在它将在其上运行的那台主机上有一个默认的绑定地址, 用不着进行这种绑定。但如果那台主机有多个地址而你又不想使用`mysqld`的默认绑定地址, 就需要用这个选项来另行绑定一个地址了。
- `--bootstrap` 这个选项是供MySQL软件的安装脚本在第一次安装该软件时使用的。它最早出现于MySQL 3.22.10版本。
- `--chroot=dir_name` 或 `-r dir_name` 运行MySQL服务器并把目录`dir_name`作为它的根目录。有关这一问题的详细讨论请参见`chroot()`函数的UNIX使用手册页。这个选项最早出现于MySQL 3.22.2版本。
- `--concurrent-insert` (布尔) 允许MyISAM数据表上的并发插入操作, `--skip-concurrent-insert`选项不允许这样做。如果MyISAM数据表里没有空洞, 并发插入操作将在你对数据表里的现有数据行进行检索操作时把新记录添加到数据表的末尾。这个布尔型选项最早出现于MySQL 4.0.2版本。在MySQL 3.23.25到4.0.1版本里只有`--skip-concurrent-insert`选项可供使用。
- `--core-file` 如果使用了这个选项, 那么, 当发生致命错误时, MySQL服务器将在退出前生成一个内核文件。这个选项最早出现于MySQL 3.23.23版本。
- `--datadir=dir_name` 或 `-h dir_name` MySQL数据目录的路径名。
- `--default-table-type=type` MySQL服务器默认使用的数据表存储类型。`type`必须是MySQL服务器所支持的数据表类型之一, 比如ISAM、MyISAM、HEAP、BDB或InnoDB。(这个值不区分字母的大小写情况。)如果没有给出这个选项, MySQL服务器将使用它在被编译时设定的默认类型(通常是MyISAM)。这个选项最早出现于MySQL 3.23版本。
- `--delay-key-write=val` 对MySQL服务器用来处理MyISAM数据表上的键值延迟写操作的模式进行设定。`val`的可取值有三种: 1) ON——根据每个数据表的具体情况来采取行动(即根据每个数据表在创建时使用的`DELAY_KEY_WRITE`选项值来决定是否要延迟其键值的写操作), 这是本选项的默认设置情况; 2) OFF——对任何一个MyISAM数据表都不进行键值延迟写操作; 3) ALL——对所有的MyISAM数据表都进行键值延迟写操作。注意: OFF和ALL对所有的MyISAM数据表都一视同仁, 不区分它们在创建时使用的是哪一种`DELAY_KEY_WRITE`选项值。  
因为`--delay-key-write=ALL`选项将对所有的MyISAM数据表都进行键值延迟写操作而不管它们当初是如何被创建的。所以, 为了在MyISAM数据表上获得更高的性能, 人们经常使用这个选项来启动镜像机制中的从服务器。  
这个选项最早出现于MySQL 3.23.3版本。它在3.23.11版本里被重新命名为`--delay-key-write-for-all-tables`, 但从MySQL 4.0.3版本开始又恢复为原来的名字。在3.23.11到4.0.3版本里, 可以通过`--skip--delay-key-write`选项来获得与如今的`--delay-key-write=OFF`选项同样的效果。
- `--des-key-file=file_name` 存放着供`DES_ENCRYPT()`和`DES_DECRYPT()`函数使用的DES密钥的文件的名字; 这个文件的格式可以在附录C对`DES_ENCRYPT()`函数的介绍内容里

查到。这个选项最早出现于MySQL 4.0.1版本。

- `--enable-locking` 请参见对`--external-locking`选项的介绍。这个选项最早出现于MySQL 4.0.3版本。
- `----enable-pstack` (布尔) 如果激活了这个选项, MySQL服务器就会在执行出错时把符号堆栈的内容打印出来。这个选项最早出现于MySQL 4.0.0版本。
- `--exit-info[=n]` 或 `-T[n]` 让MySQL服务器在退出时对信息进行调试。如果用`-T`选项来设定`n`值, 它们之间不允许有间隔性的空格; 如果它们之间有空格, MySQL服务器将无法对`n`值做出正确的解释。这个选项最早出现于MySQL 3.22版本。
- `--external-locking` (布尔) 在某些系统(比如Linux系统)里, 外部锁定机制(即文件系统级的锁定机制)是默认禁用的。这个选项将激活这类系统中的外部锁定机制。这个布尔型选项最早出现于MySQL 4.0.3版本; 在此之前, 需要使用`--enable-locking`或`--skip-locking`选项来激活或者禁用外部锁定机制。  
外部锁定机制只对那些仅进行读操作的数据表检查操作起作用。对于那些会修改数据表内容的操作(比如数据表修复操作), 应该在`isamchk`或`myisamchk`程序运行期间关闭MySQL服务器以避免损坏各有关数据表。(详情见本书第13章中的讨论。)数据表的检查和修复工作应该尽可能地使用`mysqlcheck`程序而不是`myisamchk`程序来完成。这是因为`myisamchk`程序将直接读取数据表文件, 所以你必须与MySQL服务器明确地进行协调; 而`mysqlcheck`程序所进行的检查和修复工作是通过MySQL服务器完成的, 你既不需要与MySQL服务器进行协调, 也不需要了解外部锁定机制。
- `--flush` 每完成一次修改操作, 就把所有的数据表“刷新”到磁盘上去(即把缓存区里的内容写到各有关磁盘文件里去)。这将大大降低因系统崩溃而导致数据表损坏的可能性, 但对系统的性能有着严重的影响。因此, 应该只在不那么稳定的系统上才使用这个选项。这个选项最早出现于MySQL 3.22.9版本。它只适用于MyISAM和ISAM数据表。
- `--init-file=file_name` 指定一个文件名, MySQL服务器将在启动时自动执行这个文件里的SQL语句。如果你给出的`file_name`参数是一个相对路径, `mysqld`就将以MySQL数据目录为起点来对它做出解释。在这个文件里, 每行只能包含一条SQL语句。
- `--language=lang_name` 或 `-L lang_name` 用指定语言向客户(程序)显示出错信息。`lang_name`参数最常见的取值是`english`(英语)或`german`(德语), 但也可以是某个用来存放语言文件的目录的绝对路径名。
- `--local-infile` (布尔) 允许或者禁用LOAD DATA LOCAL语句。在MySQL 3.23.49及以后的版本里, LOCAL功能(虽然已经出现)的默认设置是处于禁用状态。用`--local-infile`选项来启动MySQL服务器将激活服务器端的LOCAL机制。这个选项还可以用来关闭LOCAL功能(如果它正处于激活状态的话): 在MySQL 4.0.2及以后的版本里, 使用`--disable-local-infile`选项; 在此之前, 请使用`--local-infile=0`。
- `--log[=file_name]` 或 `-l[=file_name]` 激活与常规日志文件有关的日志机制。常规日志记录着关于客户连接和查询的一般性信息。如果你没有给定`file_name`参数, 这个日志的文件名

就是MySQL数据目录中的`HOSTNAME.log`，其中`HOSTNAME`是服务器主机的主机名。如果你给出的`file_name`参数是一个相对路径，`mysqld`将以MySQL数据目录为起点来对它做出解释。如果打算用`-l`选项来设定`file_name`参数，它们之间不允许有间隔性的空格；否则，`file_name`值可能无法得到正确的解释。

- `--log-bin[=file_name]` 激活与二进制变更日志有关的日志机制。如果你没有给定`file_name`参数，这类日志的文件名将是MySQL数据目录中的`HOSTNAME-bin.nnn`，其中`HOSTNAME`是服务器主机的主机名，`nnn`则是一个按1递增的序号（每创建一个新日志，`nnn`的值就增加一个1）。如果你给出的`file_name`参数是一个相对路径，`mysqld`将以MySQL数据目录为起点来对它做出解释。这个选项最早出现于MySQL 3.23.14版本。
- `--log-bin-index=file_name` 激活二进制变更日志索引文件。如果你给出的`file_name`参数是一个相对路径，`mysqld`将以MySQL数据目录为起点来对它做出解释。这个选项最早出现于MySQL 3.23.15版本。
- `--log-slow-queries=[file_name]` 激活慢查询日志机制且把有关信息记录到给定的文件里去。如果你没有给定`file_name`参数，这个日志的文件名将是MySQL数据目录中的`HOSTNAME-slow.log`，其中`HOSTNAME`是服务器主机的主机名。如果你给出的`file_name`参数是一个相对路径，`mysqld`将以MySQL数据目录为起点来对它做出解释。这个选项最早出现于MySQL 3.23.9版本。
- `--log-isam=[file_name]` 激活索引文件日志机制。这只用于对ISAM / MyISAM操作进行调试的场合。如果你没有给定`file_name`参数，这个日志的文件名将是MySQL数据目录中的`myisam.log`。
- `--log-long-format` 把辅助性信息写到变更日志和慢查询日志里去。这个选项最早出现于MySQL 3.22.7版本。
- `--log-update=[file_name]` 激活与变更日志文件有关的日志机制。变更日志记录着每一个对数据表内容做出了修改的查询命令的文本。如果你没有给定`file_name`参数，这类日志的文件名将是MySQL数据目录中的`HOSTNAME.nnn`，其中`HOSTNAME`是服务器主机的主机名，`nnn`则是一个按1递增的序号（每创建一个新日志，`nnn`的值就增加一个1）。如果你给出的`file_name`没有后缀名，MySQL服务器将像刚才介绍的那样给它加上一个`nnn`形式的后缀名。如果你给出的`file_name`有后缀名，MySQL服务器将不加任何修改地把它用做变更日志的文件名。如果你给出的`file_name`参数是一个相对路径，`mysqld`将以MySQL数据目录为起点来对它做出解释。

从最早出现二进制变更日志的MySQL 3.23.14版本开始，变更日志已逐渐被淘汰了。

- `--log-warnings`（布尔） 把一些非关键性警告信息写到日志文件里去。这个选项最早出现于MySQL 3.23.40版本，它那时叫做`--warnings`；`--log-warnings`是从4.0.3版本开始使用的新名字。
- `--low-priority-updates` 给修改操作分配比检索操作更低的优先级。这个选项最早出现于MySQL 3.23版本。（在从MySQL 3.22.5到MySQL 3.23版本里，这个选项叫做`--low-`



priority-inserts。)

- `--memlock` (布尔) 锁定内存中的服务器。这个选项最早出现于MySQL 3.23.25版本。这个选项只在Solaris系统上起作用,且要求MySQL服务器必须以root用户运行。
- `--myisam-recover=level` 这个选项控制着MySQL服务器在启动时自动进行的MyISAM数据表检查工作的类型。*level*可以为空,即禁用这种检查机制;也可以是以逗号分隔的一个或多个选项值:DEFAULT(与没给出任何选项的情况等价);BACKUP(给修改过的数据表创建一个备份);FORCE(即使可能造成多个数据行丢失,也要强行进行恢复);QUICK(快速恢复)。这个选项最早出现于MySQL 3.23.25版本。
- `--new` 或 `-n` 使用新的、但可能不太完善的例程。它们是MySQL软件中稳定性尚未得到最终肯定的试验性功能。如果你不想冒险,那就最好不要使用这个选项。
- `--old-protocol` 或 `-o` 使用MySQL 3.21以前的版本所使用的客户/服务器通信协议。如果你的MySQL服务器需要与一些非常早期的客户(程序)进行通信,就需要使用这个选项。
- `--old-passwords` MySQL 4.1及以后的版本使用了一种新的口令加密方法。口令仍按老方法加密的账户在新版本里仍能使用,但新口令却都是用新方法加密的。这个选项将强制MySQL服务器继续使用老方法来加密新口令。(如果你想让新版MySQL服务器支持老口令或者想把账户转移到某个老版本的服务器上,就需要使用这个选项。)这个选项最早出现于MySQL 4.1版本。
- `--one-thread` 只使用一个线程来运行MySQL服务器,用于Linux系统上的调试工作(Linux系统通常至少使用3个线程)。这个选项最早出现于MySQL 3.22.2版本。
- `--pid-file=file_name` 在启动的时候,mysqld会把自己的进程ID(即PID)写到一个文件里。这个选项给出的就是那个PID文件的路径名。其他进程可以通过这个文件来确定MySQL服务器的进程ID——通常是为了向它发出一个信号。比如说,当mysql.server脚本需要向MySQL服务器发出一个关机信号时,它就会先去读取这个文件。这个选项在Windows系统或者嵌入式MySQL服务器上没有效果。如果*file\_name*是一个相对路径,mysqld就将以MySQL数据目录为起点来对它做出解释。
- `--safe-mode` 这个选项类似于`--skip-new`选项,但将禁用更多的功能。如果MySQL运行得不太稳定,或者复杂查询的结果看起来不太正确,就应该试试这个选项。如果使用这个选项改善了MySQL服务器的操作情况,请在使用mysqlbug脚本报告你所遇到的问题时把这一点也写清楚。
- `--safe-show-database` (布尔) 如果某用户没有访问数据库的权限,则不把数据库的名字显示给该用户。这个选项最早出现于MySQL 3.23.30版本。从MySQL 4.0.2版本开始,因为新增了SHOW DATABASES权限(这个权限控制着哪些用户能够看到哪些数据库)的缘故,`--safe-show-database`选项已逐渐被淘汰。
- `--safe-user-create` (布尔) 如果某用户不具备user权限表的写权限,则不允许该用户创建新的账户。这个选项最早出现于MySQL 3.23.41版本。
- `--safemalloc-mem-limit=n` 模拟内存短缺的情况。这个选项给可分配内存量设定了一个上



限。这个选项只能用在MySQL服务器在编译阶段的配置工作中使用了--with-defug=full选项的场合。这个选项最早出现于MySQL 3.23.28版本。

- --skip-concurrent-insert 参见对--concurrent-insert选项的介绍。
- --skip-grant-tables 或 -Sg (布尔) 不使用权限表来验证客户连接。这将允许任何客户(程序)去做任何事情,它还将禁用GRANT和REVOKE语句。如果想让MySQL服务器开始使用权限表来验证客户连接,可以发出一条FLUSH PRIVILEGES语句或者发出一条mysqladmin flush-privileges命令。

这个选项的-Sg形式从MySQL 4.0.0版本开始不再允许使用。

- --skip-host-cache 禁止使用主机名缓存区。
- --skip-locking 参见对--external-locking选项的介绍。这个选项最早出现于MySQL 4.0.3版本。
- --skip-name-resolve 不对主机名进行解析。如果你使用了这个选项,就必须在权限表里把主机名写成数字形式的IP地址或者localhost。
- --skip-networking 不允许TCP/IP连接;只允许本地客户(程序)连接MySQL服务器,并且在建立连接的时候必须使用UNIX套接字并使用localhost作为主机名。
- --skip-new 不使用新的、可能不太完善的例程。参见对--new选项的介绍。
- --skip-safemalloc 不进行内存分配检查。这个选项只能用在MySQL服务器在编译阶段的配置工作中使用了--with-defug=full选项的场合。这个选项最早出现于MySQL 3.23.37版本。
- --skip-show-database 不允许无相应权限的用户发出SHOW DATABASES查询或者在他们无权访问的数据库上发出SHOW TABLES查询。这个选项最早出现于MySQL 3.23版本。
- --skip-stack-trace 在执行出错时不打印堆栈跟踪信息。这个选项最早出现于MySQL 3.23.38版本。
- --skip-symlink 不允许使用数据表的符号链接机制。这个选项最早出现于MySQL 3.23.39版本。
- --skip-thread-priority 在一般情况下,数据修改操作(即会改变数据表内容的查询)的优先级要高于数据检索操作。如果这不是你所希望的,就需要使用这个选项让MySQL服务器不给不同类型的查询赋予不同的优先级。
- --sql-bin-update-same (布尔) 把SQL\_LOG\_BIN和SQL\_LOG\_UPDATE变量合二为一。此后,只要你使用SET语句设置了它们当中的一个,另一个也将同时得到设置。这个选项最早出现于MySQL 3.23.16版本。
- --sql-mode=flags 这个选项将改变MySQL服务器的某些行为,使它符合ANSI SQL标准或者与老版本的服务器保持兼容。flags是以逗号分隔的一个或者多个下列选项:
  - ANSI\_QUOTES 把双引号(")解释为标识符(比如数据库、数据表、数据列的名字)使用的引号字符而不是供字符串使用的引号字符。反引号(`)仍允许继续用做各种名字的引号字符。

- **IGNORE\_SPACE** 允许函数名与参数表的左括号之间有空格。这等于是把函数名当做保留字。
- **NO\_UNSIGNED\_SUBTRACTION** 从MySQL 4开始，在整数减法运算中，只要两个操作数中有一个是无符号整数，其结果就将是无符号整数。这个选项将使上述运算的结果是带符号整数——与MySQL 4之前的行为保持兼容。
- **ONLY\_FULL\_GROUP\_BY** 通常，MySQL允许SELECT语句所选取的输出列不出现在GROUP BY子句里，比如下面这条语句中的数据列b：

```
SELECT a, b, COUNT(*) FROM t GROUP BY a;
```

但如果你使用了ONLY\_FULL\_GROUP\_BY标志，SELECT语句所选取的输出列就必须出现在GROUP BY子句里，如下所示：

```
SELECT a, b, COUNT(*) FROM t GROUP BY a, b;
```

- **PIPES\_AS\_CONCAT** 把“||”用做字符串合并操作符而不是逻辑OR操作符。
- **REAL\_AS\_FLOAT** 把REAL数据列类型当做FLOAT的同义词而不是DOUBLE的同义词。
- **SERIALIZE** 把SERIALIZABLE设定为默认的事务隔离级别。

类似于--ansi选项，--sql-mode选项也可以用来激活MySQL服务器的ANSI行为，但后者在使用上更加灵活——因为你可以有选择地激活或者禁用某项功能。尤其是NO\_UNSIGNED\_SUBTRACTION行为，它只能通过--sql-mode选项来设置。

--sql-mode选项最早出现于MySQL 3.23.41版本。NO\_UNSIGNED\_SUBTRACTION标志值最早出现于MySQL 4.0.0版本。

- **--temp-pool (布尔)** 如果你使用了这个选项，MySQL服务器就将使用一组数量有限的名字来命名临时文件而不是为每个临时文件创建一个独一无二的名字。在Linux系统上，这种做法能避免某些与缓存机制有关的问题。这个选项最早出现于MySQL 3.23.33版本；在MySQL 4.0.3及以后的版本里，这个选项是默认激活的。
- **--transaction-isolation=level** 设定默认的事务隔离级别。参数level的可取值包括READ-UNCOMMITTED、READ-COMMITTED、REPEATABLE-READ和SERIALIZABLE。这个选项最早出现于MySQL 3.23.26版本。
- **--tmpdir=dir\_name 或 -t dir\_name** 用来存放临时文件的目录的路径名。这个选项最早出现于MySQL 3.22.4版本。从MySQL 4.1版本开始，这个选项的值允许是一组目录，MySQL将以轮转方式来使用这些目录。在UNIX系统上，目录名之间要用冒号(:)来分隔；在Windows系统上，目录名之间要用分号(;)来分隔。
- **--warnings** 参见对--log-warnings的介绍。

#### 1. Windows选项

本小节中的选项只能在运行于Windows下的MySQL服务器上使用。在这些选项中，有的还允许被运行为一项Windows服务，这部分选项只适用于基于Windows NT的系统。

- `--console` (布尔) 用一个控制台窗口来显示出错信息。这个选项最早出现于MySQL 3.22.4版本。
- `--enable-named-pipe` (布尔) 在MySQL 3.23.50之前的版本里, 命名管道在支持命名管道机制的、基于Windows NT的MySQL服务器(即那些名称中有“-nt”字样的MySQL服务器)上是默认激活的。从MySQL 3.23.50开始, 命名管道机制是默认禁用的; 这个选项用来打开MySQL服务器对命名管道机制的支持。  
注意, 如果你激活了命名管道机制, 就可能在关闭MySQL服务器时遇到麻烦; 因此, 应该尽可能地避免激活命名管道机制。
- `--install` 把MySQL服务器安装为一项Windows服务并让它在Windows启动时自动运行。这个选项只适用于基于Windows NT的系统。
- `--install-manual` 把MySQL服务器安装为一项Windows服务但不让它在Windows启动时自动运行, 你必须明确地以手动方式亲自去启动之。这个选项最早出现于MySQL 3.23.44版本。这个选项只适用于基于Windows NT的系统。
- `--remove` 删除作为一项Windows服务的MySQL服务器。这个选项只适用于基于Windows NT的系统。
- `--standalone` 把MySQL服务器运行作为一个独立的程序而不是一项Windows服务。这个选项只适用于基于Windows NT的系统。
- `--use-symbolic-links` (布尔) 激活数据库目录的符号链接支持机制。这个选项最早出现于MySQL 3.23.17版本。从MySQL 4.0版本开始, MySQL服务器的数据库符号链接支持机制是默认激活的, 可以用`--skip-symlink`选项来关闭它。

## 2. BDB选项

本小节中的选项都是BDB数据表处理程序所独有的。

- `--bdb-home=dir_name` BDB主目录的路径名。如果你打算明确地设置这个选项, 它的值就应该与`--datadir`选项的值相同。这个选项最早出现于MySQL 3.23.14版本。
- `--bdb-lock-detect=val` 对BDB的死锁检测/解析模式进行设定。MySQL服务器将根据`val`值来决定需要让哪一个事务“流产”以解除死锁现象, `val`的可取值包括: YOUNGEST(最后开始的事务); OLDEST(“寿命”最长的事务); RANDOM(随机选择一个事务); DEFAULT(按默认策略选出的事务; 目前的默认策略是RANDOM)。这个选项最早出现于MySQL 3.23.15版本。
- `----bdb-logdir=dir_name` 用来存放BDB日志文件的目录的路径名。这个选项最早出现于MySQL 3.23.14版本。
- `--bdb-no-recover` 告诉BDB处理程序不要在启动时对BDB文件进行自动恢复。在某些场合, 如果BDB文件的自动恢复工作失败了, MySQL服务器在启动阶段就会退出运行; 这个选项将使MySQL服务器在这种情况下强行启动并运行。这个选项最早出现于MySQL 3.23.30版本。(注意, MySQL 3.23.16到3.23.29版本中的`--bdb-recover`选项的含义正好与此相反。)

- `--bdb-no-sync` 不以同步方式刷新BDB日志。这个选项最早出现于MySQL 3.23.14版本。
- `--bdb-shared-data` 以多进程模式启动BDB处理程序。这个选项最早出现于MySQL 3.23.29版本。
- `--bdb-tempdir=dir_name` 用来存放BDB处理程序临时文件的目录的路径名。这个选项最早出现于MySQL 3.23.14版本。
- `--skip-bdb` 禁用BDB数据表处理程序——如果你用不着BDB数据表，就没有必要启动BDB处理程序，这个选项能替你节省一些内存。它最早出现于MySQL 3.23.15版本。

### 3. InnoDB选项

本小节中的选项都是InnoDB数据表处理程序所独有的。大多数InnoDB选项都最早出现于MySQL 3.23.29版本，它们当时的名字都是以`--innobase`开头的；从MySQL 3.23.37版本开始，它们才被重新命名为以`--innodb`开头。

- `--innodb_data_file_path=filespec_list` InnoDB表空间组件文件的定义。*filespec\_list*的格式在本书第11章讨论InnoDB配置指令的有关内容里有详细的介绍。这个选项最早出现于MySQL 3.23.37版本。
- `--innodb_data_home_dir=dir_name` 用来存放InnoDB表空间组件文件的目录的路径名。这个选项最早出现于MySQL 3.23.37版本。
- `--innodb_fast_shutdown` (布尔) 加快MySQL服务器的关机过程，InnoDB处理程序将跳过它平时要进行的一些操作。这个选项最早出现于MySQL 3.23.44版本。
- `--innodb_flush_log_at_trx_commit=n` 这个选项的默认值是1，其含义是在你提交事务时刷新InnoDB日志（即把有关信息写到磁盘上去）。把这个选项设置为0将减少InnoDB处理程序对磁盘的写操作次数，但同时却加大了系统崩溃时丢失一些最近被提交的事务的可能性。*n*的可取值有以下几种：

取 值	含 义
0	每隔一秒写一次日志，同时刷新相应的磁盘文件
1	每提交一次事务写一次日志，同时刷新相应的磁盘文件
2	每提交一次事务写一次日志，但每隔一秒刷新一次相应的磁盘文件

这个选项最早出现于MySQL 3.23.37版本；选项值2最早出现于MySQL 3.23.52版本。

- `--innodb_flush_method=val` 这个选项只能用在UNIX系统上；它设定了InnoDB用来刷新日志的方法。这个选项的可取值是`fdatasync`（这是它的默认值）和`O_DSYNC`。这个选项最早出现于MySQL 3.23.39版本。
- `--innodb_log_group_home_dir=dir_name` 用来存放InnoDB日志文件的目录的路径名。这个选项最早出现于MySQL 3.23.37版本。
- `--innodb_log_arch_dir=dir_name` 这个选项目前尚未使用。它最早出现于MySQL 3.23.37版本。



- `--innodb_log_archive=n` 这个选项目前尚未使用。它最早出现于MySQL 3.23.37版本。
- `--skip-innodb` 禁用InnoDB数据表处理程序——如果你用不着InnoDB数据表，就没有必要启动InnoDB处理程序。这个选项能替你节省一些内存。这个选项最早出现于MySQL 3.23.37版本，它当时叫做`--skip-innobase`，`--skip-innodb`是从3.23.37版本开始使用的新名字。

#### 4. 与镜像机制有关的选项

本小节中的选项都是MySQL的镜像机制所独有的。

对于镜像机制中的从服务器，有几个选项必须与主信息文件（默认为MySQL数据目录中的`master.info`文件）配合使用。在从服务器启动的时候，如果主信息文件不存在，从服务器将使用`--master-host`、`--master-user`、`--master-password`、`--master-port`和`--master-connect-retry`等几个选项的值来建立与主服务器的连接并把这些选项值保存到主信息文件里去；如果主信息文件存在，从服务器就将使用该文件的内容而忽略上述选项。也就是说，如果你想让上述选项起作用，就必须先删除主信息文件，然后再重新启动从服务器。主信息文件的名称可以通过`--master-info-file`选项加以改变。

- `--abort-salve-event-count=n` 这个选项是`mysql-test-run`脚本用来测试镜像机制的工作情况用的。它最早出现于MySQL 3.23.28版本。
- `--binlog-do-db=db_name` 供镜像机制中的主服务器使用，含义是只记录并处理指定数据库上的数据修改操作，其他数据库将不参加镜像机制。如果你想记录并处理多个数据库上的数据修改操作，就必须使用多个`--binlog-do-db`选项来依次指定每一个数据库。这个选项最早出现于MySQL 3.23.23版本。
- `--binlog-ignore-db=db_name` 供镜像机制中的主服务器使用，含义是不记录并处理指定数据库上的数据修改操作。如果你想不记录并处理多个数据库上的数据修改操作，就必须使用多个`--binlog-ignore-db`选项来依次指定每一个数据库。这个选项最早出现于MySQL 3.23.23版本。
- `--disconnect-slave-event-count=n` 这个选项是`mysql-test-run`脚本用来测试镜像机制的工作情况用的。它最早出现于MySQL 3.23.28版本。
- `--init-rpl-role=val` 表明这个MySQL服务器在镜像机制中的角色，`val`的可取值是`master`或`salve`。这个选项是`mysql-test-run`脚本用来测试镜像机制的工作情况用的。这个选项最早出现于MySQL 4.0.0版本。
- `--log-slave-updates`（布尔） 这个选项将使镜像机制中的从服务器把接收自主服务器的变更情况记录到它自己的二进制变更日志里去。这个选项使这个从服务器还能够充当另一个从服务器的主服务器——使你能够用多个服务器构成一个镜像链。这个选项最早出现于MySQL 3.23.17版本。
- `--master-connect-retry=n` 供镜像机制中的从服务器使用，如果与主服务器的连接没有成功，则等待`n`秒后再进行重试。这个选项最早出现于MySQL 3.23.15版本。
- `--master-host=host_name` 供镜像机制中的从服务器使用，负责给定主服务器正在其上运



行的那台主机（也就是从服务器将要去连接的主机）。*host\_name*可以是一个主机名，也可以是一个IP号。这个选项最早出现于MySQL 3.23.15版本。

- `--master-info-file=file_name` 供镜像机制中的从服务器使用，负责给定保存着当前镜像状态信息的文件的名称。这个文件的内容包括：镜像机制中的主二进制变更日志的文件名和读位置；主服务器所在的主机；用来连接主服务器的用户名、口令、端口号；连接重试间隔时间等。这个文件的默认名是MySQL数据目录里的master.info文件。这个选项最早出现于MySQL 3.23.15版本。
- `--master-password=pass_val` 供镜像机制中的从服务器使用，负责给定用来连接主服务器的那个账户的口令。如果没有给出口令，则以空字符串为默认值。这个选项最早出现于MySQL 3.23.15版本。
- `--master-port=port_num` 供镜像机制中的从服务器使用，负责给定用来连接主服务器的TCP/IP端口。如果没有给出端口号，则以编译在MySQL服务器程序内部的端口号为默认值（通常是3306）。这个选项最早出现于MySQL 3.23.15版本。
- `--master-retry-count=n` 供镜像机制中的从服务器使用，负责给定主服务器的连接重试次数；如果在经过这么多次重试之后仍未成功地连接上主服务器，则放弃本次操作。这个选项最早出现于MySQL 3.23.43版本。
- `--master-ssl`（布尔）  
`--master-ssl-cert=file_name`  
`--master-ssl-key=file_name`

这些选项最早出现于MySQL 4.0.0版本，但有关功能目前还没有实现。它们是为未来能够使用安全化连接来建立镜像机制而预留的。（顺便说一句，这组选项目前并不完整——至少还需要把`--master-ssl-ca`和`--master-ssl-cipher`选项也实现出来。）

- `--master-user=user_name` 供镜像机制中的从服务器使用，负责给定用来连接主服务器的那个账户的用户名。如果没有给出用户名，则以test为默认值。这个选项最早出现于MySQL 3.23.15版本。
- `--max-binlog-dump-events=n` 这个选项是mysql-test-run脚本用来测试镜像机制的工作情况用的。这个选项最早出现于MySQL 3.23.40版本。
- `--old-rpl-compat`（布尔）告诉MySQL服务器在遇到LOAD DATA语句时使用早期的镜像二进制变更日志格式，这种格式不把数据也保存到日志里。这个选项最早出现于MySQL 4.0.0版本。
- `--reckless-slave` 这个选项是供镜像机制的调试工作使用的。这个选项最早出现于MySQL 4.0.2版本。
- `--relay-log=file_name` 供镜像机制中的从服务器使用，负责给定中继日志的文件名。（在MySQL 4里，从服务器的I/O线程负责把从主服务器那里读到的数据变更情况写到中继日志里去，再由SQL线程负责从中继日志里读出数据变更情况并完成相应的数据更新操作。）这个文件的默认名是MySQL数据目录里的HOSTNAME-relay-bin.nnn，其中HOSTNAME是服务器主机的主机名，nnn则是一个按1递增的序号（每创建一个新日志，nnn的值就增加

一个1)。这个选项最早出现于MySQL 4.0.2版本。

- `--relay-log-index=file_name` 供镜像机制中的从服务器使用，负责给定中继日志索引文件的文件名。这个文件的默认名是MySQL数据目录里的`HOSTNAME-relay-bin.index`，其中`HOSTNAME`是服务器主机的主机名。这个选项最早出现于MySQL 4.0.2版本。
- `--relay-log-info-file=file_name` 供镜像机制中的从服务器使用，负责给定中继日志信息文件的文件名。这个文件的默认名是MySQL数据目录里的`relay-log.info`。这个选项最早出现于MySQL 4.0.2版本。
- `--replicate-do-db=db_name` 供镜像机制中的从服务器使用，含义是只镜像指定数据库上的数据修改操作，其他数据库将不参加镜像机制。如果你想镜像多个数据库上的数据修改操作，就必须使用多个`--replicate-do-db`选项来依次指定每一个数据库。这个选项最早出现于MySQL 3.23.16版本。
- `--replicate-do-table=db_name.tbl_name` 供镜像机制中的从服务器使用，含义是只镜像指定数据表上的数据修改操作，其他数据表将不参加镜像机制。如果你想镜像多个数据表上的数据修改操作，就必须使用多个`--replicate-do-table`选项来依次指定每一个数据表。这个选项最早出现于MySQL 3.23.28版本。
- `--replicate-ignore-db=db_name` 供镜像机制中的从服务器使用，含义是不镜像指定数据库上的数据修改操作。如果你想不镜像多个数据库上的数据修改操作，就必须使用多个`--replicate-ignore-db`选项来依次指定每一个数据库。这个选项最早出现于MySQL 3.23.16版本。
- `--replicate-ignore-table=tbl_name` 供镜像机制中的从服务器使用，含义是不镜像指定数据表上的数据修改操作。如果你想不镜像多个数据表上的数据修改操作，就必须使用多个`--replicate-ignore-table`选项来依次指定每一个数据表。这个选项最早出现于MySQL 3.23.28版本。
- `--replicate-rewrite-db=master_db->slave_db` 这个选项告诉镜像机制中的从服务器要把主服务器中的`master_db`数据库上的修改操作刷新到从服务器中的`slave_db`数据库上。这个选项最早出现于MySQL 3.23.26版本。如果你打算在命令行上给出这个选项，就应该把该选项的值用引号引起来以避免命令解释器把“>”字符当做一个输出重定向操作符来对待。
- `--replicate-wild-do-table=pattern` 供镜像机制中的从服务器使用，含义是只对名字与给定模式相匹配的数据表进行镜像。如果你想使用多个匹配模式，就必须使用多个`--replicate-wild-do-table`选项来依次给出每一个模式。这个选项最早出现于MySQL 3.23.28版本。
- `--replicate-wild-ignore-table=pattern` 供镜像机制中的从服务器使用，含义是不对名字与给定模式相匹配的数据表进行镜像。如果你想忽略多个匹配模式，就必须使用多个`--replicate-wild-ignore-table`选项来依次给出每一个模式。这个选项最早出现于MySQL 3.23.28版本。
- `--server-id=n` 镜像机制中的主服务器的ID值。这个值在镜像机制所涉及的各个服务器当中必须是独一无二的。这个选项最早出现于MySQL 3.23.26版本。
- `--show-slave-auth-info` (布尔) 在`SHOW SLAVE STATUS`语句的输出报告里显示从服务

器的用户名和口令。这个选项最早出现于MySQL 4.0.0版本。

- `--skip-slave-start` 不自动启动从服务器；必须使用SLAVE START语句以手动方式来启动它。这个选项最早出现于MySQL 3.23.26版本。
- `--slave-load-tmpdir=dir_name` 从服务器用来处理LOAD DATA语句的目录的路径名。这个选项最早出现于MySQL 4.0.0版本。
- `--slave-skip-errors=error_list` 如果在执行过程中发生了`error_list`里列出的错误，从服务器将忽略之而不是挂起镜像进程。如果`error_list`参数的值是all，则将忽略所有错误，这与使用了`--reckless-slave`选项时的行为是一样的；否则，`error_list`参数的值就应该是以逗号分隔的一个或者多个出错代号。这个选项最早出现于MySQL 3.23.47版本。
- `--sporadic-binlog-dump-fail` (布尔) 这个选项是mysql-test-run脚本用来测试镜像机制的工作情况用的。这个选项最早出现于MySQL 3.23.40版本。

## E.12.4 与mysqld有关的变量

下面这条命令能让你查看到各mysqld变量的默认值：

```
% mysql --help
```

下面这条命令能让你查看到mysqld程序当前正使用着的变量及其取值：

```
% mysqladmin variables
```

mysqld变量的当前值还可以用SHOW VARIABLES语句来查看；能够用这条语句来查看的各有关变量可以在附录D的SHOW语句条目下查到。mysqld变量都可以按本附录前面内容里的“设置程序变量”小节所介绍的步骤进行设置。此外，MySQL 4.0.3及以后的版本还允许你对一些变量进行动态设置，这方面的详细情况请参见附录D里的SET语句条目。

## E.13 mysqld\_multi

mysqld\_multi脚本大大简化了在同一台主机上运行多个mysqld服务器的工作，既可以用它来启动或停止某个特定的MySQL服务器，也可以用它来查看它们的运行情况。

### E.13.1 用法

```
mysqld_multi [ options ] command server_list
```

`command`参数的值应该是start、stop或report之一；`server_list`参数值是你打算对之进行操作的服务器的名字。mysqld\_multi脚本的具体用法请参见本书第11章中的有关内容。

### E.13.2 mysqld\_multi支持的标准选项

```
--help          --password  --user          --version
```

当你使用mysqld\_multi脚本来停止某个特定的MySQL服务器或者查看它是否在运行时，

mysqld\_multi脚本将把它自己的--user和--password选项值传递给mysqladmin程序。

### E.13.3 mysqld\_multi独有的选项

- `--config-file=file_name` 一个选项文件名，mysqld\_multi脚本将使用这个文件里的选项来操纵MySQL服务器。如果没有使用--config-file选项，mysqld\_multi脚本将去读/etc/my.cnf文件和主目录中的.my.cnf文件以获得服务器选项。（mysqld\_multi脚本会到标准选项文件里去读取它自己的选项；--config-file选项不改变这一行为。）
- `--example` 显示一份选项文件样本，它演示了适用于mysqld\_multi脚本的各种选项文件组的用法。
- `--log=file_name` mysqld\_multi脚本用来记录其操作情况的日志文件的名称。如果这个文件已经存在，新日志信息将被追加到这个文件的末尾。默认的日志文件是/tmp/mysqld\_multi.log。可以用--no-log选项禁用此项日志功能。
- `--mysqladmin=file_name` mysqladmin程序所在的目录的路径名，mysqld\_multi脚本将调用mysqladmin程序来完成有关的操作动作。如果mysqld\_multi脚本无法找到mysqladmin程序，或者你想让它使用某个特定版本的mysqladmin程序，就需要使用这个选项。
- `--mysqld=file_name` mysqld程序所在的目录的路径名，mysqld\_multi脚本将调用mysqld程序来完成有关的操作动作。如果mysqld\_multi脚本无法找到mysqld程序，或者你想让它使用某个特定版本的mysqld程序，就需要使用这个选项。（也可以用这个选项来给出mysqld\_safe或safe\_mysqld脚本的路径名。）
- `--no-log` 显示日志输出而不是把它写到日志文件里去。（如果想查看mysqld\_multi脚本的日志输出内容，就必须使用这个选项，因为默认行为是把它们写到日志文件里去。）
- `--tcp-ip` 在默认情况下，mysqld\_multi脚本将试图使用一个UNIX套接字文件来连接MySQL服务器；这个选项将使mysqld\_multi脚本使用TCP/IP去建立这种连接。如果某个正在运行中的MySQL服务器的套接字文件已经被删除，就只能通过TCP/IP去访问它——而这需要你使用这个选项。

## E.14 mysqld\_safe

mysqld\_safe是一个用来启动和监控mysqld服务器的shell脚本；如果mysqld服务器意外“死亡”，mysqld\_safe将重新启动之。mysqld\_safe脚本在MySQL 4版本之前的名字叫safe\_mysqld。mysqld\_safe是shell脚本，在Windows系统上不可用。

### E.14.1 用法

```
mysqld_safe [ options ]
```

### E.14.2 mysqld\_safe独有的选项

能够与mysqld程序一起使用的选项都可以与mysqld\_safe脚本一起使用——后者将简单地把



这些选项传递给前者。此外，mysqld\_safe脚本还有以下一些独有的选项：

- `--core-file-size=n` 如果服务器发生崩溃，这个选项能够把内核文件的尺寸限制在 $n$ 个字节。这个选项最早出现于MySQL 3.23.28版本。
- `--err-log=file_name` 把这个文件用做错误日志。相对路径名将以mysqld\_safe脚本所在的目录为起点进行解释。如果你没有给出这个选项，默认的错误日志将是MySQL数据目录中的`HOSTNAME.err`，`HOSTNAME`是当前主机的主机名。这个选项最早出现于MySQL 3.23.22版本。
- `--ledir=dir_name` mysqld\_safe脚本将到这个目录（即所谓的“libexec”目录）里去寻找MySQL服务器。这个选项最早出现于MySQL 3.23.22版本。
- `--open-files-limit=n` 把可供使用的打开文件描述符的个数设定为 $n$ 。这个选项最早出现于MySQL 3.23.41版本。
- `--mysqld=file_name` 把`file_name`用做mysqld程序的路径。这个选项最早出现于MySQL 3.23.30版本。
- `--timezone=tzspec` 把时区设置为`tzspec`。如果你的MySQL服务器无法自动确定系统的时区，这个选项就派上用场了。这个选项最早出现于MySQL 3.23.28版本。

### E.14.3 升级注意事项

在MySQL 4.x之前的MySQL 3.x版本里，mysqld\_safe脚本的名字是safe\_mysqld，所以在把MySQL从3.x版本向4.x版本升级的时候，需要注意以下两件事：

- 如果你在老版本的启动脚本里调用了safe\_mysqld，别忘了在升级后把它们全都改为调用mysqld\_safe；mysql.server脚本就是一个例子。不过，如果你是用4.x版本去替换老版本的，那些标准的启动脚本就都应该自动改为调用mysqld\_safe了；但如果你有一些自行编写的启动脚本，千万不要忘记对它们做出相应的修改。
- 别忘了把MySQL选项文件中的[safe\_mysqld]选项组改名为[mysqld\_safe]。

## E.15 mysqldump

mysqldump程序能够把数据表的内容写到文本文件里去。它生成的那些文件可以用于许多目的，比如作为数据库备份、把数据库转移到另一个服务器去、根据现有数据库的内容建立一个供测试工作使用的数据库等等。

在默认情况下，用mysqldump程序导出的数据表将输出为一个文件，这个文件由一条CREATE TABLE语句（用来创建数据表）和紧随其后的一组INSERT语句（用来加载数据表内容）构成。但如果你使用了--tab选项，mysqldump程序将生成两个文件：数据表的内容将以纯数据格式写到一个数据文件里，而用来创建数据表的SQL语句将被写到另外一个文件里。

### E.15.1 用法

mysqldump程序有以下三种运行模式：



```
mysqldump [ options ] db_name [ tbl_name ] . . .
mysqldump [ options ] --databases db_name . . .
mysqldump [ options ] --all-databases
```

在第一种模式里，mysqldump程序将对指定数据库里的指定数据表进行转储；如果没有给出数据表的名字，mysqldump将依次对数据库里的所有数据表进行转储。在第二种模式里，mysqldump程序将把所有的参数都解释为数据库的名字，它将依次转储各指定数据库里的所有数据表。在第三种模式里，mysqldump程序将依次转储所有数据库里的所有数据表。

下面是mysqldump程序最常见的使用方法：

```
% mysqldump --opt db_name > backup_file
```

注意，用mysqldump程序导出的备份文件是不能用mysqlimport程序重新导入到MySQL里的，只能使用mysql程序来导入之，如下所示：

```
% mysql db_name < backup_file
```

### E.15.2 mysqldump支持的标准选项

--character-sets-dir	--host	--socket
--compress	--password	--user
--debug	--pipe	--verbose
--default-character-set	--port	--version
--help	--set-variable	

--character-sets-dir和--default-character-set选项是从MySQL 3.23.15版本开始新增加的。从MySQL 4开始，mysqldump程序还支持各种标准的SSL选项。

### E.15.3 mysqldump独有的选项

下面介绍的各个选项都是用来控制mysqldump程序的操作行为的。在随后的第E.15.4节里，我们将对那些用来配合--tab选项以决定数据文件格式的选项做集中的介绍。

- --add-drop-table (布尔) 在每条CREATE TABLE语句的前面加上一条DROP TABLE IF EXIST语句。这个选项最早出现于MySQL 3.22.4版本。
- --add-locks (布尔) 在用来加载各数据表内容的各组INSERT语句的前后分别加上LOCK TABLE和UNLOCK TABLE语句。这个选项最早出现于MySQL 3.22.3版本。
- --all 或 -a (布尔) 把一些附加信息（比如数据表的类型、AUTO\_INCREMENT数据列的起始编号值等等）添加到mysqldump程序所生成的CREATE TABLE语句里。这些信息构成了CREATE TABLE语法中的*table\_options*部分（另请参见附录D中的有关内容）。这个选项最早出现于MySQL 3.22.23版本。
- --all-databases 或 -A (布尔) 转储所有数据库中的所有数据表。这个选项最早出现于MySQL 3.23.12版本。
- --allow-keywords (布尔) 允许数据列的名字是MySQL关键字。这个选项最早出现于MySQL 3.22.3版本。

- `--complete-insert` 或 `-c` (布尔) 在INSERT语句里列出所有将被插入的数据列的名字。
- `--databases` 或 `-B` (布尔) 把mysqldump程序的所有参数都解释为数据库的名字, 依次转储各指定数据库里的所有数据表。这个选项最早出现于MySQL 3.23.12版本。
- `--delayed-insert` (布尔) 在转储文件里使用INSERT DELAYED语句。这个选项最早出现于MySQL 3.22.15版本。
- `--disable-keys` 或 `-K` (布尔) 在mysqldump程序的输出内容里添加ALTER TABLE ... DISABLE KEYS和ALTER TABLE ... ENABLE KEYS语句。等你重新导入这个备份文件的时候, MySQL将在处理INSERT语句时禁用索引键值的自动更新机制; 这将加快MyISAM数据表上的索引创建工作。这个选项最早出现于MySQL 3.23.48版本。
- `--extended-insert` 或 `-e` (布尔) 在转储文件里使用同时插入多个数据行的INSERT语句, 这要比使用只插入单个数据行的INSERT语句更有效率。这个选项最早出现于MySQL 3.22.15版本。
- `--first-salve` 或 `-x` (布尔) 用FLUSH TABLES WITH READ LOCK语句锁定所有数据库中的所有数据表。(注意: 这个选项并不足以使InnoDB数据表在备份过程中保持不变。换句话说, 这个选项并不能让你获得一份稳定的InnoDB数据表备份, 需要用`--single-transaction`选项来转储InnoDB数据表。) 这个选项最早出现于MySQL 3.23.22版本。
- `--flush-logs` 或 `-F` (布尔) 在转储工作开始之前先刷新MySQL服务器的日志文件(即把缓存在内存中的日志信息写到磁盘上的日志文件里去)。
- `--force` 或 `-f` (布尔) 不管是否出错都继续执行。
- `--lock-tables` 或 `-l` (布尔) 在转储各有关数据表之前必须先把它们都锁定住。这个选项不能与`--single-transaction`选项一起使用。
- `--master-data` 这个选项将使mysqldump程序所生成的备份文件能够用在镜像机制的从服务器上。它将把数据表在主服务器上的文件名和位置记载到转储文件的末尾。使用`--master-data`选项将自动激活`--first-salve`选项。这个选项最早出现于MySQL 3.23.48版本。
- `--no-autocommit` (布尔) 把对应于每一个数据表的全体INSERT语句生成为一个事务。与将在自动提交模式里执行这些INSERT语句的做法相比, 使用如此生成的转储文件将使你能够更有效率地完成数据加载工作。这个选项最早出现于MySQL 3.23.48版本。
- `--no-create-db` 或 `-n` (布尔) 不在转储文件里写出CREATE DATABASE语句。(通常, 当你使用了`--databases`或`--all-databases`选项时, mysqldump程序就会自动地在转储文件里加上一些CREATE DATABASE语句。) 这个选项最早出现于MySQL 3.23.12版本。
- `--no-create-info` 或 `-t` (布尔) 不在转储文件里写出CREATE TABLE语句。如果只想转储数据表里的数据, 就需要使用这个选项。
- `--no-data` 或 `-d` (布尔) 不在转储文件里写出数据表的内容。如果只想转储各有关数据表的CREATE TABLE语句, 就需要使用这个选项。
- `--opt` 加快数据表转储操作的速度并生成一份能加快数据表导入操作速度的转储文件来。这个选项将自动激活以下选项(具体情况还要由你的mysqldump程序的版本来决定): --

add-drop-table、--add-locks、--all、--disable-keys、--extended-insert、--lock-tables和--quick。这个选项最早出现于MySQL 3.22.3版本。

- --quick 或 -q (布尔) mysqldump程序的默认行为是先把一个数据表的内容全部读到内存里，然后再把它写出去。这个选项将使mysqldump在从MySQL服务器读到一个数据行之后立刻把它写到转储文件里去，这就大大减少了内存占用量。但需要提醒大家注意的是，如果使用了这个选项，就不应该让mysqldump程序在执行中途被挂起来——那将使MySQL服务器进入等待状态，从而影响到其他客户（程序）的正常工作。
- --quote-names 或 -Q (布尔) 把数据表和数据列的名字用反引号(`)字符引起来。如果这些名字里有MySQL保留字或者包含有特殊字符，这个选项就有用了。这个选项最早出现于MySQL 3.23.6版本。
- --result-file=file\_name 或 -r file\_name 把输出写到指定文件里去。这个选项是为基于Windows的系统准备的，它将防止出现一个换行符被自动转换为一个回车符加上一个换行符的现象。这个选项最早出现于MySQL 3.23.28版本。
- --single-transaction (布尔) 这个选项能够让InnoDB数据表（也适用于其他类型的数据表）在备份过程中保持不变，使你能够获得一份一致的InnoDB数据表备份。这一做法的关键在于它是在同一个事务里来转储各有关数据表的——这相当于同时给各有关数据表都加上了读锁定。这个选项最早出现于MySQL 4.0.2版本。它不能与--lock-tables选项一起使用。
- --tab=dump\_dir 或 -T dump\_dir 这个选项将使mysqldump程序为每个数据表生成两个转储文件并以dump\_dir参数给定的目录（这个目录必须是已经存在的）作为这些文件的存放场所：对于一个名为tbl\_name的数据表，mysqldump程序将把其中的数据保存到dump\_dir/tbl\_name.txt文件里去，把与之对应的CREATE TABLE语句保存到dump\_dir/tbl\_name.sql文件里去。此外，你还必须拥有相应的FILE权限才能使用这个选项。在默认情况下，数据文件中的每一行将以换行符结束，各数据列值之间以制表符分隔。这一格式可以通过第E.15.4节里介绍的选项来加以改变。

如果不了解--tab选项的工作情况，就很容易把它的使用效果弄混淆。下面是使用--tab选项时必须注意的两件事：

- 有些文件将被写在MySQL服务器主机上，而另一些文件则将被写在客户主机上。\*.txt文件将被写到MySQL服务器主机的dump\_dir目录里，而\*.sql文件则将被写到客户主机的dump\_dir目录里。如果这两个主机不同，mysqldump生成的转储文件就将被分别创建在不同的机器里。因此，为避免让这些文件分散在不同的机器上，最好只在服务器主机上用--tab选项来运行mysqldump程序。
- \*.txt文件的属主将是你用来运行MySQL服务器程序的那个账户，而\*.sql文件的属主则是你用来运行mysqldump程序的那个账户。造成这一现象的原因是：\*.txt文件是由MySQL服务器亲自写出来的，其内容是数据表里的数据；而\*.sql文件则是由mysqldump程序写出来的，其内容是由MySQL服务器发送至mysqldump程序的CREATE TABLE语句。
- --tables 改写--databases选项。这个选项最早出现于MySQL 3.23.12版本。

- `--where=where_clause` 或 `-w where_clause` 只对数据表中符合 `where_clause` 参数给定的 WHERE 子句的记录进行转储；这个子句应该用适当的引号引起来，以免 shell 把它错误地解释为多个命令行参数。这个选项最早出现于 MySQL 3.22.7 版本。
- `--xml` 或 `-X` 生成 XML 格式的输出生。这个选项只能用在 MySQL 3.23.51 及以后的版本里（它其实最早出现于 MySQL 3.23.48 版本，但最初的输出格式并不完善）。

#### E.15.4 mysqldump 程序的数据格式选项

如果你使用了 `--tab` 选项，mysqldump 程序就会为每一个数据表分别生成一个数据文件。此时，还可以使用以下几个额外的选项来进一步控制数据文件的格式。注意：可能需要使用适当的引号字符把下面这些选项的值引起来。这些选项与 LOAD DATA 语句所使用的数据格式选项很相似，详细情况请参见本书附录 D 中的 LOAD DATA 语句条目。

- `--fields-enclosed-by=char` 用给定字符 `char`（它通常是一个引号字符）来括住各数据列的值。这个选项的默认值是空字符，即不使用任何字符来括住各数据列的值。这个选项不能与 `--fields-optionally-enclosed-by` 选项同时使用。
- `--fields-escaped-by=char` 把给定字符 `char` 用做转义字符，即以它来开始对特殊字符进行转义的字符序列。这个选项的默认值是空，即不设定转义字符。
- `--fields-optionally-enclosed-by=char` 用给定字符 `char`（它通常是一个引号字符）来括住各非数值数据列的值。这个选项的默认值是空字符，即不使用任何字符来括住各非数值数据列的值。这个选项不能与 `--fields-enclosed-by` 选项同时使用。
- `--fields-terminated-by=str` 在数据文件里，用给定字符串 `str` 来分隔各数据列的值。它可以是一个或者多个字符。默认的数据列值分隔符是制表符。
- `--lines-terminated-by=str` 在数据文件里，用给定字符串 `str` 来分隔各输出行。它可以是一个或者多个字符。默认的输出行分隔符是换行符。这个选项最早出现于 MySQL 3.22.4 版本。

#### E.15.5 与 mysqldump 有关的变量

下面这些与 mysqldump 程序有关的变量可以按照本附录前面内容里的“设置程序变量”小节中所介绍的步骤进行设置。

- `max_allowed_packet` mysqldump 程序与 MySQL 服务器进行通信时使用的缓冲区的最大尺寸。
- `net_buffer_length` mysqldump 程序与 MySQL 服务器进行通信时使用的缓冲区的初始尺寸。这个缓冲区可以扩张到 `max_allowed_packet` 个字节长。

#### E.16 mysqlhotcopy

mysqlhotcopy 工具程序能够高效率地完成数据库和数据表的备份工作。它只能作用在 MyISAM 和 ISAM 数据表上。mysqlhotcopy 是一个 Perl 脚本，它要求你的系统必须安装有 DBI 支



持。它目前还不能用在Windows系统上。

mysqlhotcopy程序的工作流程是这样的：1) 连接本地主机上的MySQL服务器；2) 向服务器发出一系列数据表刷新和锁定语句以锁定各有关数据表；3) 把数据表文件拷贝到另一个地方作为备份。这将确保：1) 把尚未写入磁盘的数据修改情况写到磁盘上去；2) 在备份数据表的过程中，服务器将不会对数据表做出新的修改。（也就是说，mysqlhotcopy脚本实现了我们在本书第13章中描述的协议——当你直接对数据表文件进行操作的时候，服务器将不去“打扰”你正在处理的数据表。）

mysqlhotcopy程序最早出现于MySQL 3.23.11版本。

### E.16.1 用法

这个程序的使用方法有好几种。它的基本语法如下所示：

```
mysqlhotcopy [ options ] db_name[./regex/] [ new_db_name | dir_name ]
```

我们来看几个例子。下面这条命令将对数据库`db_name`进行备份，如果备份成功，在MySQL数据目录下就会增加一个名为`db_name_copy`的备份数据库：

```
% mysqlhotcopy [ options ] db_name
```

而下面这条命令将把数据库`db_name`拷贝到/tmp目录下一个名字仍为`db_name`的下级目录里去：

```
% mysqlhotcopy [ options ] db_name /tmp
```

mysqlhotcopy程序的在线文档里有更多的例子，可以用下面这条命令来查看它们：

```
% perldoc mysqlhotcopy
```

### E.16.2 mysqlhotcopy支持的标准选项

```
--debug      --host      --port      --user
--help       --password  --socket
```

--host选项（如果给出的话）只能用来指定本地主机的名字。在默认情况下，mysqlhotcopy脚本将使用一个UNIX套接字文件来连接本地主机上的MySQL服务器。如果你用--host选项给出了服务器的实名，mysqlhotcopy脚本就将使用TCP/IP来建立连接；此时，还可以用--port选项来另行指定一个非默认的端口号。--host选项最早出现于MySQL 3.23.52版本。

### E.16.3 mysqlhotcopy独有的选项

- `--allowold` 如果目标目录已经存在，在制作新备份之前先用`_old`后缀重新命名它。此后，如果备份工作失败，就要把那个目录的名字再改回来；如果备份工作成功，则删除那个目录——除非你还同时使用了`--keepold`选项。
- `--checkpoint=db_name.tbl_name` 把一个检查点记录写到指定的数据表里去，该数据表必须事先用如下所示的语句创建好：



```
CREATE TABLE tbl_name
(
    time_stamp TIMESTAMP NOT NULL,
    src          VARCHAR(32),
    dest         VARCHAR(60),
    msg          VARCHAR(255)
);
```

src和dest是源数据库和目标数据库的名字，msg是一条表明备份操作是否成功的消息。

- **--dryrun 或 -n** “不执行”模式。mysqlhotcopy脚本将报告它都会采取哪些操作动作，但并不会真正地执行这些动作。可以利用这个选项来验证mysqlhotcopy脚本的行为是不是与你想像的一样，这为你学习它的使用方法提供了方便。
- **--flushlog** 在各有关数据表全都被锁定之后、在开始拷贝它们之前，先把缓存在内存里的日志信息写到磁盘上去。这相当于在拷贝操作开始之前对它们都进行一次检查点检查。
- **--keepold** 如果目标目录已经存在，在制作新备份之前先用\_old后缀重新命名它。此后，如果备份工作失败，就要把那个目录的名字再改回来；但在备份工作成功时不删除那个目录。这个选项隐含着--allowold选项。
- **--method=copy\_method** 用来拷贝文件的方法。如果copy\_method参数的值是cp，则使用cp程序。现时期，还有一种实验性质的scp方法可供选用。在使用scp方法的时候，copy\_method参数值必须是你将使用的scp命令的完整内容，而且用来存放备份文件的目标目录必须已经存在。因为通过网络来拷贝文件需要额外花费一些时间，所以scp方法会导致各有关数据表处于锁定状态的时间大大延长。
- **--noindices** 不拷贝索引文件。如果今后需要使用如此备份出来的文件去恢复数据表，可以使用myisamchk --recover（适用于MyISAM数据表）或isamchk --recover（适用于ISAM数据表）命令去重建索引。
- **--quiet 或 -q** 只有在执行出错时才产生提示性输出信息。
- **--record\_log\_pos=db\_name.tbl\_name** 在开始拷贝数据表之前，先发出SHOW MASTER STATUS和SHOW SLAVE STATUS语句并把它们的执行结果记录到指定的数据表里去，该数据表必须在事先用如下所示的语句创建好：

```
CREATE TABLE tbl_name
(
    host          VARCHAR(60) NOT NULL,
    time_stamp    TIMESTAMP NOT NULL,
    log_file      VARCHAR(32) NULL,
    log_pos       INT NULL,
    master_host   VARCHAR(60) NULL,
    master_log_file VARCHAR(32) NULL,
    master_log_pos INT NULL,
    PRIMARY KEY (host)
);
```

SHOW MASTER STATUS语句的执行结果将被记录在log\_file和log\_pos数据列里，它们提

供了主服务器上的二进制日志的同步信息——如果备份主机是镜像机制中的主服务器，用它的备份文件初始化出来的从服务器将从这些同步信息所指定的位置开始去进行下一次镜像同步工作。SHOW SLAVE STATUS语句的执行结果将被记录在master\_host、master\_log\_file和master\_log\_pos数据列里——如果备份主机是镜像机制中的从服务器而你打算用备份文件去对同一主服务器的另一个从服务器进行初始化，就需要用到这些信息了。

- `--regex=pattern` 只对名字与给定的正则表达式相匹配的那些数据库进行拷贝。如果使用了这个选项，mysqlhotcopy命令行上的最后一个参数就必须是你将用来存放备份数据库的那个目录的名字。
- `--resetmaster` 在所有数据表被锁定之后、在开始拷贝它们之前，先发出一条RESET MASTER语句对二进制变更日志进行重置。
- `--resetslave` 在所有数据表被锁定之后、在开始拷贝它们之前，先发出一条RESET SLAVE语句对master.info文件中的信息进行重置。
- `--suffix=str` 如果你打算把备份数据库与原始数据库保存在同一个目录里，就需要使用这个选项给备份数据库的名字加上一个后缀以区分二者。新数据库目录名字的前半截与原始数据库目录的名字相同，后半截则是你用这个选项给出的字符串。
- `--tmpdir=dir_name` 用来存放临时文件的目录的路径名。这个选项的默认值是环境变量TMPDIR所指定的目录；如果该变量没有定义，则将使用/tmp作为临时目录。

## E.17 mysqlimport

mysqlimport程序是一个批量数据的加载工具，它可以把文本文件的内容读到现有的数据表里去。它的功能相当于SQL语句LOAD DATA的一个命令行接口，是一种能够非常高效地把数据行加载到数据表里去的工具。

mysqlimport程序只能读取数据文件，它不能读取mysqldump程序生成的SQL格式的转储文件，SQL格式的转储文件要用mysql程序来读取。

### E.17.1 用法

```
mysqlimport [ options ] db_name file_name . . .
```

mysqlimport程序将把数据加载到db\_name参数给出的数据库所包含的数据表里，这个数据表将由文件名参数file\_name确定。对于每一个文件名，从第一个句点开始的后半部分将被截去，剩余部分将被视为一个数据表名，而数据就将被加载到这个数据表里去。比如说，mysqlimport程序将把president.txt文件的内容加载到president数据表里去。

### E.17.2 mysqlimport支持的标准选项

<code>--character-sets-dir</code>	<code>--host</code>	<code>--socket</code>
<code>--compress</code>	<code>--password</code>	<code>--user</code>
<code>--debug</code>	<code>--pipe</code>	<code>--verbose</code>

```
--default-character-set    --port                --version
--help                    --silent
```

--character-sets-dir和--default-character-set选项分别始见于MySQL 3.23.21和MySQL 3.23.41版本。从MySQL 4开始，mysqlimport程序还支持各种标准的SSL选项。

### E.17.3 mysqlimport独有的选项

下面介绍的各个选项都是用来控制mysqlimport程序的操作行为的。在随后的第E.17.4节里，我们将对那些用来表明输入文件格式的选项做集中的介绍。

- **--columns=col\_list** col\_list是由数据表里的数据列名构成的清单，数据文件里的数据将依次被加载到这些数据列里去，而没有在col\_list里出现的数据列将被设置为它们的默认值。在col\_list里，数据列名之间要用逗号分隔开。这个选项最早出现于MySQL 3.23.17版本。
- **--delete 或 -d** (布尔) 在把数据加载到每一个数据表里之前，先彻底清除数据表里的现有内容。
- **--force 或 -f** (布尔) 不管执行是否出错，都继续加载数据。
- **--ignore 或 -i** 如果输入行里的某个值会导致数据表的惟一化索引出现重复的键值，则保留现有的数据行而丢弃来自数据文件的输入行。--ignore与--replace选项是互相排斥的，不能同时使用。
- **--ignore-lines=n** 忽略（不加载）数据文件的前n个输入行。这个选项的用途之一是跳过数据文件开头部分的数据列标题行。这个选项最早出现于MySQL 4.0.2版本。
- **--local 或 -L** (布尔) 在默认情况下，mysqlimport程序会让MySQL服务器去读取数据文件，这意味着数据文件必须存放在服务器主机上，并且你还必须具备相应的FILE权限。如果使用了--local选项，mysqlimport程序就将亲自读取数据文件并把读到的数据发送给MySQL服务器。后一种方式的数据加载速度比较慢，但它的优势是：1) 允许在服务器主机以外的其他机器上运行mysqlimport程序；2) 即便是在服务器主机上，也不要求你必须具备相应的FILE权限。

这个选项最早出现于MySQL 3.22.15版本。从MySQL 3.23.49版本开始，如果MySQL服务器被配置成不允许使用LOAD DATA LOCAL语句的情况，这个选项将没有效果。

- **--lock-tables 或 -l** (布尔) 在把数据加载到各有关数据表里去之前，先锁定之。
- **--low-priority** (布尔) 使用优先级修饰符LOW\_PRIORITY往数据表里加载数据。这个选项最早出现于MySQL 3.22.27版本。
- **--replace 或 -r** (布尔) 如果输入行里的某个值会导致数据表的惟一化索引出现重复的键值，则用来自数据文件的输入行替换掉现有的数据行。--ignore与--replace选项是互相排斥的，不能同时使用。

### E.17.4 mysqlimport程序的数据格式选项

在默认情况下，mysqlimport程序将假设数据文件中的每一行以换行符结束，各数据列值之

间以制表符分隔。但这种预期可以通过以下选项加以改变。你可能需要使用适当的引号字符把下面这些选项的值引起来。这些选项与LOAD DATA语句所使用的数据格式选项很相似，详细情况请参见本书附录D中的LOAD DATA语句条目。

- `--fields-enclosed-by=char` 表明各数据列的值是用给定字符`char`（它通常是一个引号字符）来括住的。这个选项的默认值是空字符，即没有使用任何字符来括住各数据列的值。这个选项不能与`--fields-optionally-enclosed-by`选项同时使用。
- `--fields-escaped-by=char` 表明给定字符`char`是转义字符，即它是对特殊字符进行转义的字符序列中的第一个字符。这个选项的默认值是空，即没有设定转义字符。
- `--fields-optionally-enclosed-by=char` 表明各非数值数据列的值是用给定字符`char`（它通常是一个引号字符）来括住的。这个选项的默认值是空字符，即没有使用任何字符来括住各非数值数据列的值。这个选项的默认值是空字符，即不使用任何字符来括住各非数值数据列的值。这个选项不能与`--fields-enclosed-by`选项同时使用。
- `--fields-terminated-by=str` 在数据文件里，各数据列的值是用给定字符串`str`来分隔的。它可以是一个或者多个字符。默认的数据列值分隔符是制表符。
- `--lines-terminated-by=str` 在数据文件里，各输入行是用给定字符串`str`来分隔的。它可以是一个或者多个字符。默认的输入行分隔符是换行符。

## E.18 mysql\_install\_db

`mysql_install_db`脚本能够完成以下工作：在MySQL服务器上创建数据目录、对容纳着各种权限表的mysql数据库进行初始化、创建一个空白的test数据库。`mysql_install_db`脚本将在权限表里创建几个基本的用户账户，比如root账户和各种匿名账户等。本书的第11章对这些基本账户做了详细的讨论，第11章还对如何利用口令来加强MySQL数据库系统安全的问题做了一些探讨。

`mysql_install_db`是一个shell脚本，所以它不能用在Windows系统上。从另一角度讲，基于Windows的MySQL数据库系统也不需要用到这个脚本，因为Windows版MySQL软件里的mysql和test数据库都已经预初始化好了。

### E.18.1 用法

```
mysql_install_db [ options ]
```

### E.18.2 mysql\_install\_db独有的选项

在这一小节里提到的版本号指的是允许你在命令行上使用有关选项的最早版本，可它们当中其实有很多是允许用在更早的MySQL版本里的——只要在全局级选项文件中的[mysqld]选项组里使用一些相应的设置项，就能对这些选项的值进行设定。从MySQL 3.23.29版本开始，这个脚本在启动时还会去读取[mysql\_install\_db]选项组，这就使mysql\_install\_db脚本所独有而mysqld程序却不知道其含义的选项（比如`--ldata`和`--force`）用起来更方便了。

- `--basedir=dir_name` 把这个目录用做MySQL基目录。
- `--datadir=dir_name` 或 `--ldata=dir_name` 把这个目录用做MySQL数据目录。`--datadir`选项最早出现于MySQL 3.23.17版本，`--ldata`选项最早出现于MySQL 3.23.24版本。
- `--force` 强行运行，不管当前主机名能否得到确定。此时，权限表里的记录项将使用主机的IP数字地址来创建——这意味着你只能使用IP数字而不是主机名（本地主机localhost是个例外）去连接MySQL服务器。这个选项最早出现于MySQL 3.22.17版本。
- `--user=user_name` 让MySQL服务器以用户`user_name`的身份来运行。这样，哪怕你是以UNIX系统的root身份来运行mysql\_install\_db脚本的，MySQL服务器创建出来的目录和文件也都将以`user_name`参数所给定的用户为属主。这个选项最早出现于MySQL 3.23.17版本。

## E.19 mysql.server

mysql.server是一个专门用来启动或停止mysqld服务器的脚本，它通过调用mysql\_safe脚本（或者safe\_mysql脚本，这要取决于你的MySQL软件的具体版本）来完成工作。mysql.server是一个shell脚本，在Windows系统上不可用。它最早出现于MySQL 3.22.7版本。

### 用法

```
mysql.server start
mysql.server stop
```

mysql.server脚本在基于System V风格的系统平台上比较多见，它通常被安装在/etc目录下的一个启动目录中。这样，在上电开机时，作为系统引导过程的一个组成部分，系统将自动地以start参数来调用这个脚本去启动MySQL服务器；等系统关机时，作为系统关机过程的一个组成部分，系统又将自动地以stop参数来调用这个脚本去关闭MySQL服务器。当然，你也可以通过手动方式去启动或者关闭MySQL服务器——只要在命令行上给出正确的参数就行了。

## E.20 mysqlshow

可以用mysqlshow程序查知系统中有哪些数据库、每个数据库里容纳着哪些数据表、每个数据表里又有哪些数据列或索引。它相当于SQL语句SHOW的一个命令行接口。

### E.20.1 用法

```
mysqlshow [ options ] [ db_name [ tbl_name [ col_name ] ] ]
```

如果你没有在mysqlshow命令行上给出数据库名，mysqlshow将列出服务器主机上的所有数据库。如果给出了数据库名但没有给出数据表名，mysqlshow将列出数据库里的所有数据表。如果给出了数据库名和数据表名但没有给出数据列名，它将列出数据表里的所有数据列。如果给



出了所有的名字，mysqlshow将把给定数据列的描述信息显示出来。

如果最后一个参数中包含有shell通配符（“\*”或“?”），mysqlshow程序的执行输出将仅限于与该通配符相匹配的数据库、数据表或数据列。mysqlshow程序将把“\*”或“?”当做LIKE操作符所使用的SQL通配符“%”和“\_”来对待。从MySQL 3.22.26版本开始，mysqlshow程序将把出现在命令行上的“%”和“\_”字符也解释为通配符。

### E.20.2 mysqlshow支持的标准选项

--character-sets-dir	--host	--socket
--compress	--password	--user
--debug	--pipe	--verbose
--help	--port	--version

--character-sets-dir选项是从MySQL 3.23.21版本开始新增加的。从MySQL 4开始，mysqlshow程序还支持各种标准的SSL选项。

### E.20.3 mysqlshow独有的选项

- --status 或 -i（布尔） 这个选项显示的信息与使用SHOW STATUS语句查到的信息是一样的。这个选项最早出现于MySQL 3.23版本。
- --keys 或 -k（布尔） 除关于各有关数据列的描述信息外，这个选项还能让你看到关于数据表索引的描述性信息。这个选项只有在mysqlshow命令行上给出了数据表名参数的时候才有意义。

## E.21 safe\_mysqld

safe\_mysqld是mysqld\_safe脚本在MySQL 4之前的版本里的名字。它的使用方法请参见前面内容里对mysqld\_safe脚本的介绍。在介绍mysqld\_safe脚本的时候，我们还提到了从MySQL 3.x升级到MySQL 4.x时的两个注意事项。



## 附录F C API指南

本附录对MySQL C客户程序开发库所提供的C语言应用程序设计接口进行了介绍。这个API由一组用来与MySQL服务器进行通信和访问数据库的函数以及一组供这些函数使用的数据类型构成。这个客户程序开发库里的函数可以划分为以下几大类：

- 连接管理类函数，用来建立和断开与MySQL服务器的连接。
- 出错报告类函数，用来获得出错代码和出错信息。
- 查询构造与执行类函数，用来构造数据库查询命令并把它们发送给MySQL服务器去执行。
- 结果集处理类函数，用来对从MySQL服务器返回的查询结果数据进行处理。
- 信息收集类函数，用来收集关于客户程序、MySQL服务器、协议版本以及当前连接的信息。
- 管理类函数，用来控制MySQL服务器的操作行为。
- 线程类函数，用来编写多线程客户程序。
- 与嵌入式MySQL服务器libmysqld进行通信的函数。
- 调试类函数，用来获得各种调试信息。
- 正逐渐被淘汰的函数，即那些现在已经被认为是过时了的函数。

如无特别说明，本附录所介绍的函数都至少从MySQL 3.22.0版本开始就已经存在于MySQL C客户程序开发库里了。

这里并不想把这篇指南写成一本教科书，所以其中收录的都是一些演示MySQL C客户程序开发库使用方法的代码片段。在本书的第6章里，大家可以看到一些完整的客户程序以及编写它们的指导意见。

### F.1 C语言程序的编译和链接

在源代码层次上，MySQL C客户程序开发库的调用接口是在mysql.h头文件里定义的。在自行编写MySQL客户程序时，源代码文件里必须有下面这条指令：

```
#include <mysql.h>
```

为了让编译器能够找到这个文件，可能需要在编译命令行上给出一个-Ipath选项，其中path是存放着各种MySQL头文件的目录的路径名。比如说，如果你已经把MySQL头文件安装在了/usr/include/mysql或者/usr/local/mysql/include目录里，那么，当你想对源代码文件my\_func.c进行编译时，就需要使用下面这样的命令：

```
% gcc -I/usr/include/mysql -c my_func.c
% gcc -I/usr/local/mysql/include -c my_func.c
```

如果你还需要用到其他的MySQL头文件，可以在mysql.h文件所在的目录里找到它们。比如说，头文件mysql\_com.h里包含着解释结果集元数据时需要用到的常数和宏，头文件errmsg.h和

mysql\_error.h里包含着解释出错代码时需要用到的常数。(注意：虽然你可能需要去看看mysql\_com.h文件里都有哪些内容，但并不需要明确地把它导入到你的源代码文件里，这件事可以让mysql\_com.h代劳。只需导入mysql\_com.h文件，你的程序就能自由地访问mysql\_com.h文件所定义的东西了。)

在目标文件层次上，这个客户程序开发库的调用接口由mysqlclient库负责提供。可以通过在链接命令行上给出-lmysqlclient选项的办法把这个库链接到你自行开发的程序里。你可能还需要使用-Lpath选项来告诉链接器应该到哪里去找这个库，其中path是用来安装mysqlclient库的目录。如下所示：

```
% gcc -o myprog my_main.o my_func.o -L/usr/lib/mysql -lmysqlclient
% gcc -o myprog my_main.o my_func.o -L/usr/local/mysql/lib -lmysqlclient
```

如果链接命令没有执行成功并给出了“unresolved symbol”(无法对符号做出解析)出错信息，就说明你还得再列举几个函数库(比如数学函数库(-lm)和zlib函数库(-lgz))供链接器去搜索。

为编译器提供头文件目录和为链接器提供链接标志是件很麻烦的事情，但在MySQL 3.23.21及以后的版本里，我们可以利用mysql\_config程序来简化这项工作。这个工具能够替我们把必要的编译或链接标志确定下来，如下所示：

```
% mysql_config --cflags
-I'/usr/local/mysql/include/mysql'
% mysql_config --libs
-L'/usr/local/mysql/lib/mysql' -lmysqlclient -lz -lcrypt -lnsl -lm
```

以上输出仅供参考，你们在自己的系统上可能会看到不同的结果。

## F.2 C API数据类型

在与MySQL服务器会话的过程中，需要用到和处理各种各样的信息，MySQL客户程序开发库所提供的数据类型就是为了表示这些信息而准备的。这些数据类型分别对应着MySQL会话连接本身、查询结果、结果集里的数据行、元数据(即同一结果集里的各有关数据列的描述性信息)等等。在下面的讨论里，术语“数据列”和“字段”所代表的含义是相同的。

### F.2.1 标量数据类型

MySQL用以下几种标量数据类型来表示非常大的整数、布尔值、字段或数据行偏移量。

- my\_bool 一种布尔类型，用来表示mysql\_change\_user()、mysql\_eof()、mysql\_thread\_init()等函数的返回值。
- my\_ulonglong 一种长整数类型，用来表示mysql\_affected\_rows()、mysql\_num\_rows()、mysql\_insert\_id()等函数所返回的数据行计数值或其他可能很巨大的数字。如果你想把一个my\_ulonglong值打印出来，应该把它转换为unsigned long类型并使用%lu格式符，如下所示：

```
printf ("Row count = %lu\n", (unsigned long) mysql_affected_rows (conn) );
```

在某些系统上，如果不像上面这样做，就无法把这种类型的值正确地打印出来；这是因为有关标准并没有对如何使用printf()来打印long long值做出规定，不同的系统有着不同的做法。不过，要是你想打印的值已经超过了unsigned long类型所能表示的最大数值( $2^{32}-1$ )，那么%lu格式符也将无法工作。

printf()函数的不同版本在超大数值的打印方面有着不同的实现方法，请大家自行查阅自己的printf()文档以了解其具体做法。比如说，某些系统允许使用%llu格式符来打印超大数值。

- **MYSQL\_FIELD\_OFFSET** 这种数据类型是为mysql\_field\_seek()和mysql\_field\_tell()函数而准备的，用来表示当前结果集的各MYSQL\_FIELD结构的偏移量。
- **MYSQL\_ROW\_OFFSET** 这种数据类型是为mysql\_row\_seek()和mysql\_row\_tell()函数而准备的，用来表示当前结果集里的各数据行的偏移量。

## F.2.2 非标量数据类型

MySQL用以下几种非标量数据类型来表示结构或数组。需要特别注意的是，MYSQL或MYSQL\_RES结构类型的任何一个实例都必须被视为“黑盒”——也就是说，只能引用结构本身而不能引用结构里的元素。MYSQL\_ROW和MYSQL\_FIELD类型没有这样的限制，可以随意访问这两种结构里的各个元素以获取查询结果集里的数据和元数据。

- **MYSQL** 这是MySQL客户程序开发库所提供的基本类型之一，用来表示连接句柄。每个连接句柄包含着客户程序与MySQL服务器之间的某一条连接的状态信息。要想与MySQL服务器进行会话，就得先用mysql\_init()函数去初始化一个MYSQL结构，然后再把它传递给mysql\_real\_connect()函数。在成功地建立起与MySQL服务器的连接之后，发出查询命令、生成结果集、获取出错信息等工作都必须通过该连接的连接句柄来进行。当你准备结束这次会话时，还得把连接句柄传递给mysql\_close()函数以关闭之；此后，这个句柄就不再有效了。
- **MYSQL\_FIELD** MySQL客户程序开发库使用MYSQL\_FIELD结构来存放结果集里的数据列的元数据，每个数据列都有一个与之对应的MYSQL\_FIELD结构。结果集里的MYSQL\_FIELD结构的个数可以调用mysql\_num\_fields()函数去查知。可以使用mysql\_fetch\_field()函数去依次访问各个字段的MYSQL\_FIELD结构，或者利用mysql\_field\_tell()和mysql\_field\_seek()函数快速前进/后退到某个MYSQL\_FIELD结构上去。了解MYSQL\_FIELD结构对表示或者解释数据行的内容有着重要的意义。下面是它的定义情况：

```
typedef struct st_mysql_field {
    char *name;
    char *org_name;
    char *table;
    char *org_table;
```

```

char *db;
char *def;
unsigned long length;
unsigned long max_length;
unsigned int flags;
unsigned int decimals;
enum enum_field_types type;
} MYSQL_FIELD;

```

MYSQL\_FIELD结构的各个成员的含义如下所示:

- **name** 结果集中的数据列的名字, 它是一个以NULL字符结尾的字符串。如果数据列是某个表达式的计算结果, 它的name值将是表达式的字符串形式。如果数据列或表达式是用一个别名而给出的, 它的name值就将是那个别名。比如说, 与下面这个查询命令相对应的各个name值将是"mycol"、"4\*(mycol+1)"、"mc"和"myexpr":

```
SELECT mycol, 4*(mycol+1), mycol AS mc, 4*(mycol+1) AS myexpr . . .
```

- **org\_name** 这个成员与name相似, 但数据列别名将被忽略。换句话说, org\_name里是数据列真正的名字。如果数据列是某个表达式的计算结果, 它的org\_name值将是一个空字符串。这个成员在MySQL 4.1之前的版本里不可用。
- **table** 数据列所在的数据表的名字, 它是一个以NULL字符结尾的字符串。如果数据表是用一个别名而给出的, 它的table值就将是那个别名。如果数据列是某个表达式的计算结果, 它的table值将是一个空字符串。比如说, 如果你发出了下面这个查询, 那么第一个数据列的table值就将是mytbl, 而第二个数据列的table值将是一个空字符串:

```
SELECT mycol, mycol+0 FROM mytbl . . .
```

- **org\_table** 这个成员与table相似, 但数据表别名将被忽略。换句话说, org\_table里是数据表真正的名字。如果数据列是某个表达式的计算结果, 它的org\_table值将是一个空字符串。这个成员在MySQL 4.1之前的版本里不可用。
- **db** 包含着数据列的数据表所在的数据库的名字, 它是一个以NULL字符结尾的字符串。如果数据列是某个表达式的计算结果, 它的db值将是一个空字符串。这个成员在MySQL 4.1之前的版本里不可用。
- **def** 数据列的默认值, 它是一个以NULL字符结尾的字符串。MYSQL\_FIELD结构的这个成员只有在结果集是通过mysql\_list\_fields()函数调用而获得的时候才会被设置, 在其他情况下的取值将是NULL。  
还可以通过发出DESCRIBE *tbl\_name*或SHOW COLUMNS FROM *tbl\_name*查询和检查结果集得到数据列的默认集。
- **length** 数据列的长度, 也就是你在创建数据表时在CREATE TABLE语句里为数据列设定的长度。如果数据列是某个表达式的计算结果, 它的length值将由表达式中的各个元素来确定。
- **max\_length** 数据列在结果集里的最长的值的长度。比如说, 如果某字符串数据列在结果



集里的值分别是"Bill"、"Jack"和"Belvidere", 它的max\_length值就将是9。

因为max\_length值只有在所有数据行都被看过之后才能确定, 所以它只对你使用mysql\_store\_result()函数而创建出来的结果集才有实际意义。对于使用mysql\_use\_result()函数创建出来的结果集, max\_length值将是0。

- flags flags成员指定数据列的属性, 每种属性对应着flags值里的一个位——可以用表F-1列出的位掩码常数来测试它们。比如说, 如果你想知道某数据列的值是不是UNSIGNED, 就要使用如下所示的测试:

```
if (field->flags & UNSIGNED_FLAG)
    printf ("%s values are UNSIGNED\n", field->name);
```

表F-1 MYSQL\_FIELD flags成员所使用的位掩码值及其含义

flags值	含 义
AUTO_INCREMENT_FLAG	数据列具备AUTO_INCREMENT属性
BINARY_FLAG	数据列具备BINARY属性
MULTIPLE_KEY_FLAG	数据列是某个非惟一化索引的组成部分
NOT_NULL_FLAG	数据列不允许包含NULL值
PRI_KEY_FLAG	数据列是某个PRIMARY KEY的组成部分
UNIQUE_KEY_FLAG	数据列是某个UNIQUE索引的组成部分
UNSIGNED_FLAG	数据列具备UNSIGNED属性
ZEROFILL_FLAG	数据列具备ZEROFILL属性

若BINARY\_FLAG标志被置位, 则表明数据列是区分字母大小写情况的, 这包括明确地用BINARY关键字定义出来的数据列(比如BINARY CHAR)以及BLOB数据列。

此外, 还有几个用来表明数据列类型而非数据列属性的flags位掩码常数, 但它们现在已经被逐渐淘汰了, 这是因为你应该用field->type来确定数据列的类型。表F-2列出了这些已逐渐被淘汰的flags位掩码常数。

表F-2 已逐渐被淘汰的MYSQL\_FIELD flags位掩码常数及其含义

flags值	含 义
BLOB_FLAG	数据列包含BLOB或TEXT值
ENUM_FLAG	数据列包含ENUM值
SET_FLAG	数据列包含SET值
TIMESTAMP_FLAG	该数据列包含TIMESTAMP值

- decimals 数值数据列的小数部分的位数, 非数值数据列的这个值是0。比如说, DECIMAL(8, 3)的decimal值是3, 而BOLB数据列的decimal值是0。
- type 数据列的类型。如果数据列是某个表达式的计算结果, 它的type值将根据表达式各个元素的类型来确定。比如说, 如果mycol是一个VARCHAR(20)数据列, 它的type值就将是FIELD\_TYPE\_VAR\_STRING, 而LENGTH(mycol)的type值却是FIELD\_TYPE\_LONGLONG。type成员的可取值见表F-3, 它们都是在mysql\_com.h文件里

定义的。

表F-3 MYSQL\_FIELD type成员的可取值

type值	含 义
FIELD_TYPE_BLOB	BLOB或TEXT
FIELD_TYPE_DATE	DATE
FIELD_TYPE_DATETIME	DATETIME
FIELD_TYPE_DECIMAL	DECIMAL或NUMERIC
FIELD_TYPE_DOUBLE	DOUBLE或REAL
FIELD_TYPE_ENUM	ENUM
FIELD_TYPE_FLOAT	FLOAT
FIELD_TYPE_INT24	MEDIUMINT
FIELD_TYPE_LONG	INT
FIELD_TYPE_LONGLONG	BIGINT
FIELD_TYPE_NULL	NULL
FIELD_TYPE_SET	SET
FIELD_TYPE_SHORT	SMALLINT
FIELD_TYPE_STRING	CHAR
FIELD_TYPE_TIME	TIME
FIELD_TYPE_TIMESTAMP	TIMESTAMP
FIELD_TYPE_TINY	TINYINT
FIELD_TYPE_VAR_STRING	VARCHAR
FIELD_TYPE_YEAR	YEAR

你也许会在某些早期的源代码文件里看到FIELD\_TYPE\_CHAR，这是一个单字节的存储类型，它现在的名字是FIELD\_TYPE\_TINY。类似地，FIELD\_TYPE\_INTERVAL现在叫做FIELD\_TYPE\_ENUM。

- **MYSQL\_RES** SELECT或SHOW查询将以结果集的形式把数据返回给客户程序，这个MYSQL\_RES结构就是用来表示各种结果集的。数据库查询命令所返回的数据行信息都包含在相应的MYSQL\_RES结构里。

在获得结果集之后，可以调用相应的API函数来提取结果集的数据部分（即结果集中各数据行的数据值）和元数据部分（关于结果集的描述性信息，比如有多少个数据列、它们的类型、它们的长度等等）。

- **MYSQL\_ROW** MYSQL\_ROW类型包含着结果集里的一个数据行的值，这些值被表示为一个字符串数组。所有的值都将被返回为字符串形式（数值也不例外）；但如果数据行中的某个值是NULL，那它在MYSQL\_ROW结构里就将被表示为一个C语言中的NULL指针。

可以用mysql\_num\_fields()函数来查知每个数据行里都有多少个值（即有多少个数据列）。第*i*个数据列的值可以通过row[i]提取出来；*i*的取值范围是0到mysql\_num\_fields(res\_set)-1，res\_set是一个指向相应的MYSQL\_RES结果集的指针。

注意，MYSQL\_ROW类型已经是一个指针了，所以应该像下面这样去定义一个row变量：

```
MYSQL_ROW row;          /* 正确 */
```

不能像下面这样做：

```
MYSQL_ROW *row;          /* 不正确 */
```

有一点请大家要特别注意：MYSQL\_ROW数组里的值都是以NULL字符结尾的，所以把各种非二进制的数值当做以NULL字符结尾的字符串来处理是不会有问题的；但如果数据值本身就包含有二进制数据，其中就很可能出现NULL字符——必须把这种数据值当做计数字符串来对待，通过它们的长度而不是NULL字符来判断其结束位置。mysql\_fetch\_lengths()函数能够让你得到一个数组，这个数组的各个元素是结果集里的某一个数据行的各个值（即各个数据列值）的长度，它的使用方法如下所示：

```
unsigned long *length;
length = mysql_fetch_lengths (res_set);
```

数据行中第*i*个数据列值的长度可以通过length[i]提取出来。如果某个数据列值是NULL，它的长度就将是0。

### F.2.3 操作符形式的宏

为了让我们能够更方便地对MYSQL\_FIELD结构中的各有关成员进行测试，mysql.h文件还定义了一些宏。IS\_NUM()对type成员进行测试，其他几个宏则对flags成员进行测试。

- IS\_NUM()——如果数据列是数值类型，则为真（非零）：

```
if (IS_NUM (field->type))
    printf ("Field %s is numeric\n", field->name);
```

- IS\_PRI\_KEY()——如果数据列是某个PRIMARY KEY的组成部分，则为真（非零）：

```
if (IS_PRI_KEY (field->flags))
    printf ("Field %s is part of primary key\n", field->name);
```

- IS\_NOT\_NULL()——如果数据列不允许包含NULL值，则为真（非零）：

```
if (IS_NOT_NULL (field->flags))
    printf ("Field %s values cannot be NULL\n", field->name);
```

- IS\_BLOB()——如果数据列是BLOB或TEXT类型，则为真（非零）。这个宏是通过测试flags成员的BLOB\_FLAG位而得出结论的；但因为BLOB\_FLAGS位已逐渐被淘汰，所以使用IS\_BLOB()宏的做法现在已经很少见了。

## F.3 C API函数

以下各节将对MySQL客户程序开发库所提供的C API函数按功能分类并做出详细的介绍，各类别中的函数将按字母表顺序排列。有些参数会在后面的讨论内容里反复出现，所以这里先对它们的含义做一个集中的说明，在介绍各有关函数的时候就不再重复了：

- conn：一个指针，它指向一个与某MySQL服务器连接相关联的MYSQL连接句柄结构。
- res\_set：一个指针，它指向一个与某SQL查询语句相关联的MYSQL\_RES结果集结构。
- field：一个指针，它指向一个MYSQL\_FIELD数据列信息结构。

- row: 一个MYSQL\_ROW数组, 即结果集中的某个数据行。
- row\_num: 结果集里的数据行的序号, 从0到mysql\_num\_row()-1 (数据行总个数减去1)。
- col\_num: 结果集里某一个数据行的数据列的序号, 从0到mysql\_num\_fields()-1 (数据列总个数减去1)。

为简洁起见, 如果后面的讨论内容在提到以上参数的时候没有对之做出特别的说明, 大家就可以把它们理解为上述含义。

### F.3.1 连接管理类函数

连接管理类函数能完成以下几项工作: 1) 建立或断开与MySQL服务器的连接; 2) 对与此有关的各种连接选项进行设置; 3) 在连接因超时而被MySQL服务器自动切断时重建连接; 4) 改变当前用户名。

在实际工作中, 最常见的做法是这样的: 先调用mysql\_init()函数去初始化一个连接句柄; 然后把这个连接句柄传递给mysql\_real\_connect()函数去建立一个连接; 最后, 当你不再需要用到这个连接的时候, 调用mysql\_close()函数来关闭之。如果还想对连接选项做些特殊的设置或者需要建立的是一条SSL连接, 就得在调用mysql\_init()函数之后、调用mysql\_real\_connect()函数之前发出一些mysql\_options()或mysql\_ssl\_set()调用。

- my\_bool

```
mysql_change_user ( MYSQL *conn,
                    const char *user_name,
                    const char *password,
                    const char *db_name );
```

在conn参数给定的连接上改变当前用户和默认数据库。如果你在用到某个数据表的时候没有指明它来自哪一个数据库, 系统就将默认它来自当前的默认数据库。如果db\_name参数的值是NULL, 则不选定默认数据库。

mysql\_change\_user()的返回值是布尔类型, 它返回真 (非零) 值的条件是: 1) 参数user\_name给定的用户有权连接MySQL服务器并连接成功; 2) 如果还给出了db\_name参数, 该用户还必须有权访问给定的数据库。对于其他情况, 这个函数将返回假 (零) 值, 当前用户和默认数据库不会改变。

与先关闭当前连接然后再使用另一套参数来重新打开一个新连接的做法相比, 用这个函数来改变当前用户和默认数据库的做法在速度上要快一些。还可以用这个函数来实现永久性连接, 使你的客户程序在执行期间能够为多个用户提供服务。

这个函数最早出现于MySQL 3.23.3版本。

- void

```
mysql_close ( MYSQL *conn );
```

关闭conn参数给定的连接。调用这个函数将结束客户程序与MySQL服务器之间的这次会话。如果连接句柄是由mysql\_init()函数自动分配的, mysql\_close()将释放之。

如果与MySQL服务器的连接没有成功，不要调用mysql\_close()。

#### • MYSQL \*

**mysql\_init ( MYSQL \*conn );**

初始化一个连接句柄并返回一个指向该句柄的指针。如果conn参数指向一个已经存在的MYSQL句柄结构，mysql\_init()将对它进行初始化并返回它的地址，如下所示：

```
MYSQL conn_struct, *conn ;
conn = mysql_init (&conn_struct );
```

如果conn参数是NULL，mysql\_init()将分配一个新句柄(指为新句柄分配内存空间等资源)、初始化之、再返回它的地址，如下所示：

```
MYSQL *conn ;
conn = mysql_init ( NULL );
```

如有可能，应该尽量使用第二个办法，少用第一个办法。把连接句柄的资源分配和初始化工作都交给客户库去完成有很多好处。比如说，如果你对MySQL软件进行了升级而新版本中的MYSQL结构与老版本中的有差异，采用第二个办法就能使你避免很多与共享库有关的问题。

虽然很少发生，但如果mysql\_init()调用失败——比如当系统资源不足以让它分配一个新句柄时，它将返回NULL。

如果mysql\_init()调用分配了连接句柄，mysql\_close()调用将在你关闭这条连接时自动释放它。

这个函数最早出现于MySQL 3.22.1版本。

#### • int

**mysql\_options ( MYSQL \*conn,  
enum mysql\_option option,  
const char \*arg );**

这个函数能够对mysql\_real\_connect()函数的连接建立行为做进一步的调控。应该在调用了mysql\_init()函数之后、调用mysql\_real\_connect()函数之前去调用这个函数。如果需要设置多个选项，可以多次调用mysql\_options()函数。(如果多次调用mysql\_options()函数去设置同一个选项，mysql\_real\_connect()函数在建立连接时使用的将是最后一次给这个选项设置的值。)

option参数是你准备设置的连接选项的名字。在设置某个选项的时候，如果还需要用到其他的信息，就必须把它们放在arg参数里。(注意：arg参数必须是一个指针。)如果不需要用到其他信息，就必须把arg参数值设置为NULL。

如果选项设置成功，mysql\_options()函数将返回零；如果option参数值不合法，则将返回非零。

这个函数最早出现于MySQL 3.22.1版本，但option参数的一些可取值却是后来才增加的，具体情况见以下各有关条目里的说明。option参数目前有以下几种可取值：



- **MYSQL\_INIT\_COMMAND**

在连接成功后立刻执行一条查询命令。arg指向一个以NULL字符结尾的字符串，字符串的内容就是你准备以如此方式去执行的SQL语句。注意：这条SQL语句在这个连接每次重建成功后（比如你调用了mysql\_ping()函数的时候）也会立刻执行。注意：这个查询所返回的结果集将被丢弃。

这个选项最早出现于MySQL 3.22.10版本。

- **MYSQL\_OPT\_COMPRESS**

如果MySQL服务器支持，本次连接将使用压缩的客户/服务器协议。这个函数的arg参数是NULL。

这个选项也可以在调用mysql\_real\_connect()函数时进行设定。

- **MYSQL\_OPT\_CONNECT\_TIMEOUT**

连接动作的倒计时时间：如果在经过这么多秒之后仍未成功地连接上MySQL服务器，客户程序将放弃这次执行。这个函数的arg参数（指针）指向一个unsigned int值，倒计时的时间值就存放在这个unsigned int值里。

- **MYSQL\_OPT\_LOCAL\_INFILE**

激活或者禁用LOAD DATA LOCAL语句。这个函数的arg参数（指针）指向一个unsigned int值：如果这个值非零，则激活此项功能；如果这个值为零，则禁用之。但如果MySQL服务器被配置成不允许使用LOAD DATA LOCAL语句的情况，这个选项将没有任何效果。

这个选项最早出现于MySQL 3.23.49版本。（在MySQL 3.22.15到MySQL 3.23.48版本里，LOAD DATA LOCAL语句是默认激活的；在MySQL 3.22.15之前的版本里，这一机制不存在。）

- **MYSQL\_OPT\_NAMED\_PIPE**

要求客户程序使用命名管道来连接MySQL服务器。这个函数的arg参数是NULL。这个选项是为将在Windows系统上使用的客户程序而准备的，而且只能用来连接基于Windows NT的MySQL服务器。

这个选项最早出现于MySQL 3.22.5版本。

- **MYSQL\_READ\_DEFAULT\_FILE**

要求客户程序从给定的选项文件里读取连接参数而不使用标准选项文件里的选项。这个函数的arg参数（指针）指向一个以NULL字符结尾的字符串，其内容是一个文件名；客户程序将从这个文件的[client]选项组里读取连接选项。如果你还使用MYSQL\_READ\_DEFAULT\_GROUP选项给出了一个选项组名称，客户程序就将读取这个选项文件里的指定选项组里的选项。

这个选项最早出现于MySQL 3.22.10版本。

- **MYSQL\_READ\_DEFAULT\_GROUP**

要求客户程序从给定的选项组中读取连接参数。这个函数的arg参数（指针）指向一个以NULL字符结尾的字符串，其内容是一个选项组名称（注意：选项组名称的两端不需

要用 “[” 和 “]” 字符括起来)；客户程序将从选项文件的[client]选项组和本选项给定的选项组里读取连接选项。如果你还使用MYSQL\_READ\_DEFAULT\_FILE选项给出了一个选项文件名，客户程序将只读取选项文件里的选项；否则，客户程序将到一系列标准选项文件里去读取各有关选项组里的选项。

注意，如果你既没有给出MYSQL\_READ\_DEFAULT\_FILE选项，也没有给出MYSQL\_READ\_DEFAULT\_GROUP选项，客户程序将不读取任何选项文件。

这个选项最早出现于MySQL 3.22.10版本。

#### • MYSQL\_SET\_CHARSET\_DIR

字符集文件所在的目录的路径名。这个函数的arg参数（指针）指向一个以NULL字符结尾的字符串，其内容就是该目录的路径名。这个目录必须存在于客户主机上。这个选项的用途是这样的：有些不常用的字符集在MySQL客户程序开发库里只有定义文件，字符集文件本身并没有被编译在这个库里；当你的客户程序需要用到这样的字符集时，就需要使用这个选项来告诉它到哪里去寻找字符集文件。

这个选项最早出现于MySQL 3.23.14版本。

#### • MYSQL\_SET\_CHARSET\_NAME

客户程序使用的默认字符集的名字。这个函数的arg参数（指针）指向一个以NULL字符结尾的字符串，其内容就是该字符集的名字。

这个选项最早出现于MySQL 3.23.14版本。

在以上选项中，有些允许你通过选项文件来给出，也就是说，可以把有关选项放在某个选项文件的[client]选项组或用MYSQL\_READ\_DEFAULT\_GROUP选项给定的选项组里，再用适当的MYSQL\_READ\_DEFAULT\_FILE选项来要求客户程序去读取指定的选项文件。本书第E.1.3节对客户程序如何从各有关选项组里读取选项的问题做了比较详细的讨论。

当在基于Windows的系统上使用MYSQL\_READ\_DEFAULT\_FILE或MYSQL\_SET\_CHARSET\_DIR选项的时候，Windows路径名中的反斜线字符“\”必须写成“/”或“\\”——推荐使用“/”。

在下面的例子里，在通过一系列mysql\_options()调用对有关的连接选项做出相应的设置之后，mysql\_real\_connect()将：1) 从C:\my.cnf.extra文件中的[client]和[mygroup]选项组里读取各有关选项；2) 使用命名管道去连接MySQL服务器；3) 连接动作的倒计时时间为10秒；4) 在连接成功后立刻执行一条SET SQL\_BIG\_TABLES = 1语句；5) 客户程序与MySQL服务器在这条连接上的通信将使用压缩协议。

```
MYSQL *conn;
unsigned int timeout;

if ((conn = mysql_init (NULL)) == NULL)
    ... deal with error ...
mysql_options (conn, MYSQL_READ_DEFAULT_FILE, "C:/my.cnf.extra");
mysql_options (conn, MYSQL_READ_DEFAULT_GROUP, "mygroup");
mysql_options (conn, MYSQL_OPT_NAMED_PIPE, NULL);
timeout = 10;
```

```
mysql_options (conn, MYSQL_OPT_CONNECT_TIMEOUT, (char *) &timeout);
mysql_options (conn, MYSQL_INIT_COMMAND, "SET SQL_BIG_TABLES=1");
mysql_options (conn, MYSQL_OPT_COMPRESS, NULL);
if (mysql_real_connect (conn, ...) == NULL)
    ... deal with error ...
```

- int

**mysql\_ping ( MYSQL \*conn );**

检查conn参数所给定的连接是否仍然有效, 如果已经断开, mysql\_ping()将使用客户程序当初建立这个连接时使用的选项去重建这个连接。因此, 如果最初的mysql\_real\_connect()调用没有成功, 就不应该再发出mysql\_ping()调用。如果该连接依然有效或者成功地再次建立起这个连接, mysql\_ping()将返回零; 如果执行出错, 则将返回一个非零值。

这个函数最早出现于MySQL 3.22.1版本。

- MYSQL \*

**mysql\_real\_connect ( MYSQL \*conn,**

```
    const char *host_name,
    const char *user_name,
    const char *password,
    const char *db_name,
    unsigned int port_num,
    const char *socket_name,
    unsigned int flags );
```

连接MySQL服务器并返回一个指向该连接句柄的指针。这个函数的conn参数(指针)必须指向一个已经存在且已经用mysql\_init()初始化过的连接句柄。如果调用成功, 这个函数的返回值将是该句柄的地址; 如果执行出错, 则将返回NULL。

如果连接操作没有成功, 可以把conn传递给mysql\_errno()或mysql\_error()函数去获得出错信息, 但不能把它传递给MySQL客户程序开发库中必须以连接成功为前提才能正确执行的其他例程。

host\_name参数负责设定MySQL服务器主机的主机名。表F-4列出了将在UNIX和Windows系统上运行的客户程序对各种host\_name参数值的解释。需要特别要注意的是: 主机名"hostname"只能在基于UNIX的系统上使用, 它表示你想通过UNIX套接字而不是通过TCP/IP来建立连接。如果你想使用TCP/IP来连接一个运行在本地主机上的MySQL服务器, 就需要把host\_name参数的值设置为"127.0.0.1"(即以字符串形式表示的本地主机的回送接口的IP地址)而不能把它设置为"localhost"。

user\_name就是你的MySQL用户名。如果这个参数的值是NULL, 客户程序开发库将提供一个默认的名字: 在UNIX系统上, 这个默认的用户名就是你的登录名; 在Window系统上, 它将是环境变量USER的值——如果这个环境变量没有定义, 则使用"ODBC"作为默认值。password参数就是你的MySQL口令。如果这个参数的值是NULL, 就只能连接到允许你不使用口令去连接的数据库(即user权限表里与给定主机名和你的用户名相对应且口令字段

为空的数据库)上去, 连接其他数据库的尝试将以失败告终。

表F-4 客户程序对mysql\_real\_connect()函数的hostname参数值的解释

hostname 参数值	UNIX 连接类型	Windows 连接类型
字符串形式的主机名	使用TCP/IP连接指定主机	使用TCP/IP连接指定主机
IP数字形式的主机名	使用TCP/IP连接指定主机	使用TCP/IP连接指定主机
localhost	使用UNIX套接字连接本地主机	使用TCP/IP连接本地主机
127.0.0.1	使用TCP/IP连接指定主机	使用TCP/IP连接指定主机
. (句点)	不使用	使用命名管道连接本地主机
NULL	使用UNIX套接字连接本地主机	使用TCP/IP连接本地主机; 在基于Windows的系统上, 先尝试使用命名管道连接本地主机, 如果不成功, 再使用TCP/IP连接本地主机

db\_name参数就是你打算使用的那个数据库的名字。如果这个参数的值是NULL, 则表示客户程序将不选定初始的默认数据库。

port\_num参数是TCP/IP连接所使用的端口号。如果这个参数的值是0, 则表示客户程序将使用默认的端口号进行连接。

socket\_name参数是UNIX套接字(如果使用的是UNIX系统的话)或命名管道(如果使用的是Windows系统的话)的文件名。如果这个参数的值是NULL, 则表示客户程序将使用默认的套接字或命名管道。

port\_num和socket\_name参数值的具体含义还要取决于host\_name参数的值: 在UNIX系统上, 如果给出的host\_name参数值是"local host", mysql\_real\_connect()函数就将使用一个UNIX域的套接字来进行连接; 在Windows系统上, 如果给出的host\_name参数值是".", mysql\_real\_connect()函数就将使用一个命名管道来进行连接; 除上述两种情况外, mysql\_real\_connect()函数将使用TCP/IP来进行连接。

flags参数的值既可以是一个或者多个下列选项, 也可以是0(意思是“没有选项”)。这些选项将对MySQL服务器的运行情况产生影响:

- CLIENT\_COMPRESS

如果MySQL服务器支持, 这条连接将使用压缩的客户/服务器协议。

- CLIENT\_FOUND\_ROWS

对于UPDATE查询, MySQL服务器将返回与该查询的WHERE子句相匹配的数据行个数, 而不是被它改变的数据行个数。这个选项将导致数据修改类查询命令在经过MySQL优化器处理后执行得稍慢一些。

- CLIENT\_IGNORE\_SPACES

在默认情况下, 函数名与参数表的左括号必须紧挨着, 它们之间不允许有空格。如果使用了这个选项, MySQL服务器将忽略函数名与参数表的左括号之间的空格。这等于是把函数名当做保留字来对待。

这个选项最早出现于MySQL 3.22.7版本。

- CLIENT\_INTERACTIVE

表明这个客户程序是一个交互式客户程序。对于由交互式客户程序建立的连接，如果它在MySQL服务器变量interactive\_timeout所设定的时间（以秒为单位）内没有操作动作，MySQL服务器就可以把连接关闭掉。一般说来，interactive\_timeout变量的值与wait\_timeout变量的值相等。

这个选项最早出现于MySQL 3.23.28版本。

- CLIENT\_NO\_SCHEMA

禁用db\_name.tbl\_name.col\_name语法。如果客户程序使用了这个选项，MySQL服务器将只能识别查询命令中的tbl\_name.col\_name、tbl\_name或col\_name语法。

- CLIENT\_ODBC

表明这个客户程序是一个ODBC客户程序。

这个选项最早出现于MySQL 3.22.4版本。

- CLIENT\_SSL

这个选项仅供SSL连接内部使用，客户程序不应该使用它。

- MYSQL\_TRANSACTIONS

向MySQL服务器表明这个客户程序支持事务处理机制。也就是说，如果客户程序没有运行在自动提交模式，那么，当它与MySQL服务器之间的连接在某次事务的过程中意外断开时，MySQL服务器将自动把各有关数据表回滚为本次事务开始之前的样子，而客户程序也知道MySQL服务器会这样做。

这个选项最早出现于MySQL 3.23.17版本。

上面这些flags参数值都是位值，所以可以用二进制位操作符“|”或者“+”把它们组合在一起使用以得到叠加的效果。比如说，下面这两个表达式就是等价的：

```
CLIENT_COMPRESS | CLIENT_ODBC
CLIENT_COMPRESS + CLIENT_ODBC
```

mysql\_real\_connect()函数的db\_name参数最早出现于MySQL 3.22.0版本。调用mysql\_init()函数来初始化一个连接句柄结构的做法最早出现于MySQL 3.22.1版本。

- int \*

```
mysql_ssl_set ( MYSQL *conn,
                const char *key,
                const char *cert,
                const char *ca,
                const char *capath,
                const char *cipher );
```

这个函数用在需要建立一条通往MySQL服务器的SSL安全化连接的场合。如果OpenSSL支持机制没有被编译到客户程序开发库里，mysql\_ssl\_set()调用将什么也不做；否则，它将对mysql\_real\_connect()调用建立一条加密连接所需要的各有关参数进行设置。（换句话



说，如果你的客户程序想建立一条安全化连接，就得先调用mysql\_ssl\_set()函数、然后再调用mysql\_real\_connect()函数。)

mysql\_ssl\_set()函数调用的返回值永远是0——在SSL参数设置阶段发生的错误将导致随后的mysql\_real\_connect()报告一个执行出错。

mysql\_ssl\_set()函数的参数key是密钥文件的路径名；cert是证书文件的路径名；ca是颁证机构文件的路径名。capath是用来存放信任证书(trusted certificates，持有这类证书的主机将被客户主机认为是值得信任的)的目录的路径名，客户主机将使用这个目录里的证书来验证此后接收到的证书。cipher是一个字符串，其内容是客户主机使用的一个或者多个加密算法的名称。上述参数的值都允许是NULL，意思是不需要mysql\_ssl\_set()函数对相应的参数进行设置。

mysql\_ssl\_set()函数最早出现于MySQL 4.0.0版本。它要求你必须事先对MySQL软件做好相应的配置。(关于安全化连接的详细讨论请参见本书第12章中的有关内容。)

### F.3.2 出错报告类函数

客户程序需要通过以下函数来查知和报告有关函数调用执行出错的原因。

#### • unsigned int

**mysql\_errno (MYSQL \*conn);**

返回给定连接上最近一次被调用的MySQL C API函数的出错代码(注意，有些MySQL C API函数不返回状态信息)。如果前一个函数调用没有出错，则将返回0；否则，将返回一个非零值。出错代码的含义可以在头文件errmsg.h和mysqld\_error.h里查到。

```
if (mysql_errno (conn) == 0)
    printf ("Everything is okay\n");
else
    printf ("Something is wrong!\n");
```

#### • const char \*

**mysql\_error (MYSQL \*conn);**

返回给定连接上最近一次被调用的MySQL C API函数的出错信息字符串(注意，有些MySQL C API函数不返回状态信息)，这个字符串以NULL字符结尾。如果前一个函数调用没有出错，则将返回一个空字符串(注意：空字符串是一个零长度的字符串""，而不是一个NULL指针)。比较常见的做法是先检查前一个函数调用是否执行出错，如果出错，再调用mysql\_error()函数去查明其具体原因，但mysql\_error()函数的返回值本身也可以用来检测前一个函数调用是否执行出错，如下所示：

```
char *err = mysql_error (conn);
if (err[0] == '\0') /* empty string? */
    printf ("Everything is okay\n");
else
    printf ("Something is wrong!\n");
```

### F.3.3 查询构造与执行类函数

客户程序需要通过以下函数来构造数据库查询命令并把它们发送给MySQL服务器去执行。

如果查询命令里包含有需要特殊对待的字符,还要用mysql\_real\_escape\_string()对它们进行转义。每个查询命令字符串只能由一条SQL语句构成且不得以分号(;)或“\g”结尾。分号(;)或“\g”是mysql客户程序使用的表示方法,不适用于MySQL C客户程序开发库。

• int

**mysql\_query ( MYSQL \*conn, const char \*query\_str );**

把由参数query\_str给定的查询命令发送给MySQL服务器去执行,查询命令被表示为一个以NULL字符结尾的字符串。这个字符串不应该包含有二进制数据;具体地说,就是不应该包含有NULL字符——因为mysql\_query()函数将把它遇到的第一个NULL字符解释为该查询命令的结束标志。如果你的查询命令真的包含有二进制数据,就应该用mysql\_real\_query()函数来发送它。mysql\_real\_query()的执行速度要比mysql\_query()稍快一些。

如果执行成功,mysql\_query()将返回0;如果执行出错,则将返回一个非零值。所谓“成功的”查询指的是本身没有语法错误且在MySQL服务器上执行时也没发生执行出错的查询,“成功的”查询并不意味着肯定会有数据行受到它的影响或者会返回一些数据行。

• unsigned long

**mysql\_real\_escape\_string ( MYSQL \*conn,  
char \*to\_str,  
const char \*from\_str,  
unsigned long from\_len );**

对包含有特殊字符的字符串进行编码,使它能够在一条SQL语句里,编码操作会把当前字符集的因素也考虑在内。所谓特殊字符以及它们的编码结果见表F-5。(注意:SQL匹配模式字符“%”和“\_”并没有出现在这个表里。)

表F-5 mysql\_real\_escape\_string()字符编码方案

特殊字符	编码结果
NUL ( ASCII 0 )	\0 ( 反斜线-0 )
\ ( 反斜线 )	\\ ( 反斜线-反斜线 )
' ( 单引号 )	\' ( 反斜线-单引号 )
" ( 双引号 )	\\" ( 反斜线-双引号 )
换行符	\\n ( 反斜线-n )
回车符	\\r ( 反斜线-r )
Ctrl-Z	\\Z ( 反斜线-Z )

MySQL本身要求进行转义的字符只有出现在字符串内的反斜线字符(\\)和出现在字符串两端的引号字符(“”或“”);mysql\_real\_escape\_string()还要对其他字符进行转义的原因是:1)为了使结果字符串更容易阅读;2)为了使结果字符串在日志文件里更容易处理。将被编码的字符串保存在指针参数from\_str所指向的缓冲区里,这个字符串将被视为一个计数型字符串(counted string,即一个根据其长度而非NULL字符来判断它是否结束的字符串),其长度(以字节为单位计算)由参数from\_len给定。mysql\_real\_escape\_string()将把编码结果写到指针参数to\_str指定的缓冲区里并在其末尾加上一个NULL字符。指针参

数to\_str所指向的缓冲区至少要有(from\_len\*2)+1个字节长——在最坏的情况下，源字符串from\_str中的每一个字符都需要被编码为一个由两个字节组成的序列，并且还需要为充当结果字符串结束标志的NULL字符留出一个位置。

mysql\_real\_escape\_string()函数的返回值是编码结果字符串的长度，作为其结束标志的NULL字符不计算在内。

编码结果字符串的内部没有NULL字符，但以NULL字符结尾，这就使你能够使用strlen()或strcat()函数来处理它们。

当你在客户程序里写出字符串的时候，千万不要把C语言转义字符（即反斜线字符“\”）本身与你用mysql\_real\_escape\_string()函数编码出来的转义序列混为一谈。请看下面这段源代码以及它产生的输出：

```
to_len = mysql_real_escape_string (conn, to_str, "\0\\\''\"\\n\\r\\032", 7);
printf ("to_len = %d, to_str = %s\\n", to_len, to_str);
```

这段源代码的输出是：

```
to_len = 14, to_str = \0\\\''\"\\n\\r\\2
```

在上面的例子里，输出字符串to\_str与源代码中的mysql\_real\_escape\_string()函数的第三个参数看起来几乎一模一样，但它们在本质上却完全不同。

mysql\_real\_escape\_string()函数最早是作为mysql\_escape\_string()函数的替代而出现于MySQL 3.23.14版本里的。因为mysql\_escape\_string()函数在对字符串进行编码的时候没有把当前字符集的因素考虑进来，所以现在已经很少有人使用它了。如果你想让自己的客户程序在MySQL软件的任何版本下都能运行，请把源代码里的mysql\_real\_escape\_string()调用全部替换为如下所示的代码片段：

```
#if !defined(MYSQL_VERSION_ID) || (MYSQL_VERSION_ID<32314)
#define mysql_real_escape_string(conn,to_str,from_str,len) \
    mysql_escape_string(to_str,from_str,len)
#endif
```

这样，如果在早期的MySQL版本下来编译你的源代码，其中的mysql\_real\_escape\_string()调用就都将被映射为mysql\_escape\_string()调用了。

• int

```
mysql_real_query ( MYSQL *conn,
                  const char *query_str,
                  unsigned long length );
```

把给定查询命令发送给MySQL服务器去执行，查询命令被表示为一个计数型字符串（counted string，即一个根据其长度而非NULL字符来判断它是否结束的字符串）。这种字符串允许包含二进制数据（包括NULL字符在内）。查询命令的文本由参数query\_str给定，其长度由参数length给定。

如果执行成功，mysql\_real\_query()将返回0；如果执行出错，则将返回一个非零值。所谓“成功的”查询指的是本身没有语法错误且在MySQL服务器上执行时也没发生执行出错的查询，“成功的”查询并不意味着肯定会有数据行受到它的影响或者会返回一些数据行。

在MySQL 4之前的版本里，参数length的类型是unsigned int。

• int

**mysql\_select\_db** ( MYSQL \*conn, consr char \*db\_name );

把db\_name参数给定的数据库选定为当前的默认数据库；如果你在用到某个数据表的时候没有指明它来自哪一个数据库，系统就将默认它来自当前的默认数据库。如果你不具备访问该数据库的权限，mysql\_select\_db()函数将调用失败。

mysql\_select\_db()非常适合用来在同一个连接上切换使用不同的数据库。一般说来，应该在当初发出mysql\_real\_connect()调用的时候先选定一个初始的默认数据库，这比在连接建立起来之后再发出一个mysql\_select\_db()调用的做法要快得多。

如果调用成功，mysql\_select\_db()将返回0；如果调用失败，则将返回一个非零值。

### F.3.4 结果集处理类函数

对于那些会产生结果集的查询，本小节里的函数将使你能够把结果集检索到客户主机上并访问其中的内容。mysql\_store\_result()和mysql\_use\_result()函数负责创建结果集，在调用本小节中的其他函数之前，必须先发出一个mysql\_store\_result()或mysql\_use\_result()调用；表F-6对两个函数进行了对比。

表F-6 mysql\_store\_result()和mysql\_use\_result()的对比

mysql_store_result()	mysql_use_result()
结果集中的所有数据行将由mysql_store_result()亲自取回到客户主机上来	mysql_use_result()只对结果集进行了初始化，数据行将由后续的mysql_fetch_row()调用取回
占用的内存较多；所有的数据行都将保存在客户主机上	占用的内存较少；每次只存放一个数据行
较慢，因为需要为整个结果集分配内存	较快，因为只需为当前数据行分配内存
若mysql_fetch_row()的返回值是NULL，则表明到达结果集里的最后一个数据行，而不是出错	若mysql_fetch_row()的返回值是NULL，则表明到达结果集里的最后一个数据行或者发生一个错误——比如因网络通信故障而无法取回当前记录等
在发出mysql_store_result()调用之后，随时都能调用mysql_num_rows()函数	在结果集里的数据行全都被取回之前，即使发出mysql_num_rows()调用，也不会得到正确的数据行计数值
mysql_affected_rows()是mysql_num_rows()函数的同义词	mysql_affected_rows()无法使用
允许通过mysql_data_seek()、mysql_row_seek()和mysql_row_tell()函数以随机方式去访问结果集里的数据行	无法对结果集进行随机方式的访问；只能依次对自MySQL服务器返回的数据行进行处理。无法使用mysql_data_seek()、mysql_row_seek()和mysql_row_tell()函数
数据表的锁定时间较短；在结果集被取回之后，各有关数据表上的锁定就解除了	数据表的锁定时间较长；只有在所有的数据行都被取回之后，各有关数据表上的锁定才会解除——如果你在此期间挂起了客户程序，各有关数据表将仍处于锁定状态，从而阻塞准备对各有关数据表进行修改的其他客户程序的执行
结果集各MYSQL_FIELD结构中的max_length成员将被设置为有一个有意义的值，即各有关数据列在结果集里长度最大的那个值的长度	结果集各MYSQL_FIELD结构中的max_length成员无法被设置为有一个有意义的值，因为它只有在结果集里的数据行全都被取回之后才能确定

- **my\_ulonglong**

**mysql\_affected\_rows** ( MYSQL \*conn );

返回最近一次DELETE、INSERT、REPLACE或UPDATE查询所改变的数据行的个数。对于这类没有结果集可供返回的查询，应该在mysql\_query()调用成功之后立刻发出一个mysql\_affected\_rows()调用。当然，也可以在发出一个有结果集可供返回的查询命令后调用这个函数；此时，mysql\_affected\_rows()将与mysql\_num\_rows()有着同样的行为并受到同样的限制，比如返回值在何种情况下才有意义等（注意：如果结果集是通过发出mysql\_use\_result()调用而生成的，那么mysql\_affected\_rows()函数将不会返回任何有意义的返回值）。

如果尚未发出任何查询命令或者本该返回一些数据行的查询命令没能从数据库检索到任何数据行，mysql\_affected\_rows()的返回值将是零。如果这个返回值大于零，那么，对于DELETE、INSERT、REPLACE或UPDATE查询，它就是受其影响的数据行的个数；对于会产生结果集的查询，它就是结果集里的数据行的个数。如果这个返回值是-1，则表示mysql\_affected\_rows()调用在执行过程中出错或者你调用mysql\_affected\_rows()函数的次序不正确（比如说，你在发出一个会返回一些数据行的查询命令之后、但在把它的结果集全部检索完毕之前发出了mysql\_affected\_rows()调用的次序就不正确）。需要提醒大家注意的是：mysql\_affected\_rows()函数的返回值是一个无符号整数，如果你想判断这个返回值是否对应着一个负整数，就必须先把它转换为一个带符号整数才能进行有关比较操作，如下所示：

```
if ((long) mysql_affected_rows (conn) == -1)
    fprintf (stderr, "Error!\n");
```

如果最初的mysql\_real\_connect()调用的flags参数里使用了CLIENT\_FOUND\_ROWS选项或者做过其他类似的设置，那么，对于UPDATE查询，mysql\_affected\_rows()的返回值将是与该查询的WHERE子句相匹配的数据行的个数，而不是被它实际改变的数据行的个数。（如果某数据行各有关数据列的值在修改前后是一样的，MySQL将认为它没有被改动。）

mysql\_affected\_rows()的返回值是my\_ulonglong类型；本附录前面内容里的第F.2.1节对如何打印输出这一类型的值的方法进行了介绍。

- **void**

**mysql\_data\_seek** ( MYSQL\_RES \*res\_set,  
my\_ulonglong row\_num );

定位到结果集里的指定数据行处。row\_num参数的取值范围是0到mysql\_num\_rows(res\_set)-1；如果你给出的row\_num参数值超出了这个范围，mysql\_data\_seek()调用的执行情况将无法预料。

mysql\_data\_seek()函数只有在整个结果集全都被检索到客户主机之后才能正确执行，所以应该只在结果集是由mysql\_store\_result()而不是由mysql\_use\_result()所创建的场合里才使用它。

mysql\_data\_seek()与mysql\_row\_seek()是有区别的：前者的第二个参数被解释为数据行在



结果集里的序号，而后者的第二个参数却被解释为数据行在结果集里的偏移量（比如 `mysql_row_tell()` 调用的返回值）。

在MySQL 3.23.7之前的版本里，这个函数的 `row_num` 参数的类型是 `unsigned int`。

#### • MYSQL\_FIELD \*

**mysql\_fetch\_field** ( MYSQL\_RES \*res\_set );

返回一个MYSQL\_FIELD结构，其中包含着关于结果集里的某个数据列的描述性信息（即该数据列的元数据）。在成功地执行了一条会返回一些数据行的查询命令之后，第一个 `mysql_fetch_field()` 调用将返回关于结果集里的第一个数据列的信息，此后的 `mysql_fetch_field()` 调用将依次返回关于结果集里的下一个数据列的信息或者NULL值（当到达最后一个数据列之后）。

与本函数配合使用的其他函数包括：1) `mysql_field_tell()`，用来查知当前数据列是结果集里的第几个数据列；2) `mysql_field_seek()`，用来定位到结果集里的指定数据列，下一个 `mysql_fetch_field()` 调用将返回关于该数据列的信息。

下面这段代码将首先定位到结果集的第一个MYSQL\_FIELD结构，然后依次提取出各后续数据列的MYSQL\_FIELD结构：

```
MYSQL_FIELD      *field;
unsigned int      i;

mysql_field_seek (res_set, 0);
for (i = 0; i < mysql_num_fields (res_set); i++)
{
    field = mysql_fetch_field (res_set);
    printf ("column %u: name = %s max_length = %lu\n",
            i, field->name, field->max_length);
}
```

#### • MYSQL\_FIELD \*

**mysql\_fetch\_fields** ( MYSQL\_RES \*res\_set );

这个函数的返回值是一个数组，数组中的各个元素依次是与结果集里的各个数据列相对应的MYSQL\_FIELD结构。可以像下面这样来访问这个数组的内容：

```
MYSQL_FIELD      *field;
unsigned int      i;

field = mysql_fetch_fields (res_set);
for (i = 0; i < mysql_num_fields (res_set); i++)
{
    printf ("column %u: name = %s max_length = %lu\n",
            i, field[i].name, field[i].max_length);
}
```

请把这段代码与我们刚才在介绍 `mysql_fetch_field()` 函数时给出的例子做一下对比。请注意：虽然这两个函数所返回的值是同样的类型，但它们的内容却要用稍微不同的语法去访问。`mysql_fetch_field()` 的返回值（指针）指向一个MYSQL\_FIELD结构；而 `mysql_fetch_fields()` 的返回值（指针）指向一个以MYSQL\_FIELD结构为元素的数组。



各种数据类型（数值类型也不例外）的值都将被表示为一个字符串。如果你需要用某个值来进行数学运算，就必须亲自使用atoi()、atof()、sscanf()等函数去对之进行类型转换。

如果已经到达结果集里的最后一个数据行，mysql\_fetch\_row()将返回NULL值。如果结果集是用mysql\_use\_result()调用生成的，那么，因为mysql\_fetch\_row()函数的每次调用只能从MySQL服务器那里取回一个数据行，所以网络通信故障也会使它返回NULL值。

数据值都是以NULL字符结尾的，但如果你已经知道某个数据值包含有二进制数据，就千万不要把它当做以NULL字符结尾的字符串来对待——应该把它当做计数型字符串（即一个需要通过其长度而非NULL字符来判断它是否结束的字符串）来对待。此时，需要知道这种数据列值的长度——可以通过mysql\_fetch\_lengths()调用来完成这一任务。

请看下面这段代码，它通过一个循环来依次提取构成同一数据行的各项数据并判断它是否是NULL值：

```
MYSQL_ROW      row;
unsigned int    i;

while ((row = mysql_fetch_row (res_set)) != NULL)
{
    for (i = 0; i < mysql_num_fields (res_set); i++)
    {
        printf ("column %u: value is %s\n",
                i, (row[i] == NULL ? "NULL" : "not NULL"));
    }
}
```

数据列值的类型可以通过存放在MYSQL\_FIELD数据列信息结构里的数据列元数据来确定，可以通过mysql\_fetch\_field()、mysql\_fetch\_fields()、mysql\_fetch\_field\_direct()等调用来完成这一任务。

#### • unsigned int

**mysql\_field\_count (MYSQL \*conn);**

返回给定连接上最近一次查询所选取的数据列的个数。mysql\_field\_count()函数通常用在mysql\_store\_result()或mysql\_use\_result()调用返回了一个NULL值的场合，它能够告诉你该查询是否应该返回一个结果集：如果mysql\_field\_count()函数调用的返回值是0，说明该查询的确不会生成一个结果集，其执行过程也就没有出错；如果它返回的是一个非零值，就说明该查询本应该返回一些数据列，既然不是这样（因为mysql\_store\_result()或mysql\_use\_result()调用返回了一个NULL值），所以该查询在执行过程中肯定出错了。

下面这段代码演示了如何使用mysql\_field\_count()函数来判断数据库查询操作是否出错的具体做法：

```
res_set = mysql_store_result (conn);
if (res_set == NULL) /* no result set was returned */
{
    /*
     * does the lack of a result set mean that an error
```

```

    * occurred or that no result set should be expected?
    */
    if (mysql_field_count (conn) > 0)
    {
        /*
         * a result set was expected, but mysql_store_result()
         * did not return one; this means an error occurred
         */
        printf ("Problem processing the result set\n");
    }
    else
    {
        /*
         * a result set was not expected; query returned no data
         * (it was not a SELECT, SHOW, DESCRIBE, or EXPLAIN),
         * so just report number of rows affected by query
         */
        printf ("%lu rows affected\n",
                (unsigned long) mysql_affected_rows (conn));
    }
}
else /* a result set was returned */
{
    /* ... process rows here, then free result set ... */
    mysql_free_result (res_set);
}

```

mysql\_field\_count()函数最早出现于MySQL 3.22.24版本；在此之前的版本里，可以用mysql\_num\_fields()函数来达到同样的目的。如果你想让自己的客户程序在MySQL软件的任何版本下都能运行，请把源代码里的mysql\_field\_count()调用全部替换为如下所示的代码片段：

```

#if !defined(MYSQL_VERSION_ID) || (MYSQL_VERSION_ID<32224)
#define mysql_field_count mysql_num_fields
#endif

```

这样，如果在早期的MySQL版本下来编译你的源代码，其中的mysql\_field\_count()调用就都将被映射为mysql\_num\_fields()调用了。

#### • MYSQL\_FIELD\_OFFSET

**mysql\_field\_seek** ( MYSQL\_RES \*res\_set, MYSQL\_FIELD\_OFFSET offset );

在结果集里，定位到offset参数所指定的那个MYSQL\_FIELD数据列信息结构处，下一个mysql\_fetch\_field()调用将返回该MYSQL\_FIELD结构。注意：offset参数不是某数据列的索引，而是一个MYSQL\_FIELD\_OFFSET类型的值（比如此前的某个mysql\_field\_tell()或mysql\_field\_seek()调用的返回值）。

如果你想定位到结果集里的第一个数据列的MYSQL\_FIELD结构处，就要把offset参数的值设置为零。

- **MYSQL\_FIELD\_OFFSET**

**mysql\_field\_tell** ( MYSQL\_RES \*res\_set );

返回当前MYSQL\_FIELD数据列信息结构的偏移量。可以把这个偏移量传递给mysql\_field\_seek()调用作为其offset参数。

- **void**

**mysql\_free\_result** ( MYSQL\_RES \*res\_set );

释放结果集所占用的内存。必须调用mysql\_free\_result()函数去释放客户程序所生成的每一个结果集。结果集通常是由mysql\_store\_result()或mysql\_use\_result()调用生成的,但有些MySQL C API函数(比如mysql\_list\_dbs()、mysql\_list\_fields()、mysql\_list\_processes()、mysql\_list\_tables()等等)会隐含地生成一个结果集,这些结果集所占用的内存也需要由你来负责释放。

在MySQL 3.22.3及以后的版本里,对于mysql\_use\_result()调用生成的结果集,mysql\_free\_result()将先自动取回并丢弃其中尚未被你取回的数据行,然后再释放之。

- **my\_ulonglong**

**mysql\_insert\_id** ( MYSQL \*conn );

返回给定连接上最近一次执行的数据库查询命令所生成的AUTO\_INCREMENT值。如果尚未执行任何查询或者前一个查询根本就不生成一个AUTO\_INCREMENT值,这个函数调用将返回零。(零返回值与合法的AUTO\_INCREMENT值是有区别的,因为后者都是些正整数。)

如果你知道自己发出的数据库查询命令会生成一个新的AUTO\_INCREMENT值并想知道它到底是多少,就应该在发出这个查询命令后立刻发出一个mysql\_insert\_id()调用。如果你在发出mysql\_insert\_id()调用之前又执行了另外一条数据库查询命令,mysql\_insert\_id()的返回值就可能会发生变化。请注意:这种行为与SQL函数LAST\_INSERT\_ID()是不同的——mysql\_insert\_id()由客户端负责处理,每发出一个数据库查询命令,它的返回值就会发生相应的改变;LAST\_INSERT\_ID()由服务器端负责处理,其返回值与具体的查询命令相关联,只要你还没有使用同一个SQL语句句柄又生成一个新的AUTO\_INCREMENT值,它的返回值就保持不变。

mysql\_insert\_id()函数的返回值与某个具体的“客户-服务器”连接相关联,不会因发生在其他连接上的AUTO\_INCREMENT活动而受到影响。

mysql\_insert\_id()的返回值是my\_ulonglong类型,本附录前面内容里的第F.2.1节对如何打印输出这一类型的值的方法进行了介绍。

- **unsigned int**

**mysql\_num\_fields** ( MYSQL\_RES \*res\_set );

返回结果集里的数据列的个数。mysql\_num\_rows()函数的常见用途是对结果集里的当前数据行中的各个数据列进行遍历,如下所示:

```
MYSQL_ROW      row;
unsigned int    i;
```



```

while ((row = mysql_fetch_row (res_set)) != NULL)
{
    for (i = 0; i < mysql_num_fields (res_set); i++)
    {
        /* do something with row[i] here ... */
    }
}

```

在MySQL 3.22.24之前的版本里，mysql\_num\_fields()函数还经常被用在一些如今是通过mysql\_field\_count()函数去完成的操作里，比如对mysql\_store\_result()或mysql\_use\_result()调用是否返回了一个NULL值进行测试以判断它们的执行是否出错等等——这也正是你在老版本的源代码里经常会看到人们以一个连接句柄指针而不是以一个结果集指针作为参数去调用mysql\_num\_fields()函数的原因。在早期的版本里，mysql\_num\_fields()函数是允许人们以这两种参数形式去调用它的，但以一个连接句柄指针为参数去调用这个函数的做法现在已逐渐被淘汰了。现在，应该用mysql\_field\_count()函数去获得数据列的计数值，请参见前面对mysql\_field\_count()函数的介绍。

#### • my\_ulonglong

**mysql\_num\_rows ( MYSQL\_RES \*res\_set );**

返回结果集里的数据行的个数。如果结果集是用mysql\_store\_result()函数生成的，就可以在它调用成功之后的任何时候发出mysql\_num\_rows()调用，如下所示：

```

if ((res_set = mysql_store_result (conn)) == NULL)
{
    /* mysql_num_rows() can be called now */
}

```

如果结果集是用mysql\_use\_result()函数生成的，那么，只有在你把各有关数据行全都从MySQL服务器那里检索回来之后，mysql\_num\_rows()调用才会有一个正确的返回值，如下所示：

```

if ((res_set = mysql_use_result (conn)) == NULL)
{
    /* mysql_num_rows() cannot be called yet */
    while ((row = mysql_fetch_row (res_set)) != NULL)
    {
        /* mysql_num_rows() still cannot be called */
    }
    /* mysql_num_rows() can be called now */
}

```

mysql\_num\_rows()函数的返回值是my\_ulonglong类型，本附录前面内容里的第F.2.1节对如何打印输出这一类型的值的方法进行了介绍。

#### • MYSQL\_ROW\_OFFSET

**mysql\_row\_seek ( MYSQL\_RES \*res\_set, MYSQL\_ROW\_OFFSET offset );**

定位到结果集里的指定数据行处。`mysql_row_seek()`函数与`mysql_data_seek()`函数的功能相同,但前者的第二个参数值(即offset参数的值)并不是某数据行的序号。这个offset参数或者是一个MYSQL\_ROW\_OFFSET类型的值(比如此前的某个`mysql_row_tell()`或`mysql_row_seek()`调用的返回值),或者等于0(意思是定位到结果集里的第一个数据行处)。`mysql_row_seek()`函数的返回值是它这次调用开始执行之前的数据行偏移量。

`mysql_row_seek()`函数只有在整个结果集全都被检索到客户主机之后才能正确执行,所以应该只在结果集是由`mysql_store_result()`而不是由`mysql_use_result()`所创建的场合里才使用它。

#### • MYSQL\_ROW\_OFFSET

**mysql\_row\_tell ( MYSQL\_RES \*res\_set );**

返回结果集里的当前数据行在结果集中的位置偏移量。这个返回值不是一个数据行序号;可以把它传递给`mysql_row_seek()`,但不能把它传递给`mysql_data_seek()`。

`mysql_row_tell()`函数只有在整个结果集全都被检索到客户主机之后才能正确执行,所以应该只在结果集是由`mysql_store_result()`而不是由`mysql_use_result()`所创建的场合里才使用它。

#### • MYSQL\_RES \*

**mysql\_store\_result ( MYSQL \*conn );**

在成功地执行了一条数据库查询命令之后,把查询命令的结果集返回并保存到客户主机上。如果查询命令没有数据可供返回或者执行出错,则将返回NULL;此时,需要调用`mysql_field_count()`函数或者调用MySQL C API所提供的出错报告类函数来判断是没有结果集可供返回还是在执行过程中发生了错误。

在完成了对结果集的处理工作后,别忘了用`mysql_free_result()`函数去释放它。

另请参见表F-6对`mysql_store_result()`和`mysql_use_result()`函数的对比。

#### • MYSQL\_RES \*

**mysql\_use\_result ( MYSQL \*conn );**

在成功地执行了一条数据库查询命令之后,为查询命令初始化出一个结果集来,但不把数据行检索到客户主机上。必须通过`mysql_fetch_row()`调用来一个一个地依次取回结果集中的数据行。如果查询命令没有数据可供返回或者执行出错,则将返回NULL;此时,需要调用`mysql_field_count()`函数或者调用MySQL C API所提供的出错报告类函数来判断是没有结果集可供返回还是在执行过程中发生了错误。

在完成了对结果集的处理工作后,别忘了用`mysql_free_result()`函数去释放它。在MySQL 3.22.3及以后的版本里,对于`mysql_use_result()`调用生成的结果集,`mysql_free_result()`将自动取回并丢弃其中尚未被你取回的数据行,然后再释放之。在MySQL 3.22.3之前的版本里,你必须先亲自去取回并丢弃结果集里尚未被取回的数据行,然后再发出一个`mysql_free_result()`调用去释放之;否则,你在发出下一条查询命令时就将看到一条报告说结果集与查询命令不配套的“out of sync”出错信息。

另请参见表F-6对`mysql_store_result()`和`mysql_use_result()`函数的对比。

`mysql_store_result()`和`mysql_use_result()`函数都能用来创建结果集，但它们对其他的结果集处理类MySQL C API函数却有着不同的要求和影响。

### F.3.5 信息收集类函数

这些函数用来收集关于客户程序、MySQL服务器、协议版本以及当前连接的信息。这些信息大都是在“客户-服务器”连接建立阶段从MySQL服务器上检索出来并被保存到MySQL C客户程序开发库里的。

• `const char *`

**`mysql_character_set_name ( MYSQL *conn );`**

返回值：给定连接上的默认字符集的名字，如"latin1"。

这个函数最早出现于MySQL 3.23.21版本。

• `const char *`

**`mysql_get_client_info ( void );`**

返回值：客户程序开发库的版本号，它被表示为一个以NULL字符结尾的字符串，比如"3.23.51"。

• `const char *`

**`mysql_get_host_info ( MYSQL *conn );`**

返回值：关于给定连接的描述性信息，它被表示为一个以NULL字符结尾的字符串，比如"Localhost via UNIX socket"、"cobra.snake.net via TCP/IP"、". via named pipe"等等。

• `unsigned int`

**`mysql_get_proto_info ( MYSQL *conn );`**

返回值：一个用来表明给定连接所使用的客户/服务器协议的数字编号。

• `const char *`

**`mysql_get_server_info ( MYSQL *conn );`**

返回值：MySQL服务器的版本号，它被表示为一个以NULL字符结尾的字符串，比如"4.0.2-alpha-log"。这个版本号由版本序号和一个或者多个后缀组成。各种版本号后缀的含义可以在附录C对`VERSION()`函数的介绍内容里查到。

• `const char *`

**`mysql_info ( MYSQL *conn );`**

返回值：一个字符串，这个字符串对最近一次执行的以下几种数据库查询命令的效果进行了描述。以下内容每两行为一组，其中第一行是有关的数据库查询命令，第二行则是相应的`mysql_info()`调用所输出的字符串格式：

`ALTER TABLE ...`

`Records: 0 Duplicates: 0 Warnings: 0`

`INSERT INTO ... SELECT ...`

`Records: 0 Duplicates: 0 Warnings: 0`

`INSERT INTO ... VALUES (...), (...), ...`

`Records: 0 Duplicates: 0 Warnings: 0`

```
LOAD DATA ...
```

```
Records: 0 Deleted: 0 Skipped: 0 Warnings: 0
```

```
UPDATE ...
```

```
Rows matched: 0 Changed: 0 Warnings: 0
```

以上内容中的数字“0”将根据你具体执行的数据库查询命令而变化。

有两点需要大家注意：1) 只有同时插入两个或两个以上记录的INSERT INTO ... VALUES语句才会使mysql\_info()调用返回一个非NULL值；2) 对于那些没有出现在上面清单里的SQL语句，mysql\_info()的返回值永远是NULL。

此外，mysql\_info()所返回的字符串是用MySQL服务器当前使用的语言（比如英语、德语等等）写出来的，所以不能想当然地认为你能够通过查找某些特定单词的办法来对它做出分析和解释。

#### • const char \*

```
mysql_stat ( MYSQL *conn );
```

返回值：MySQL服务器的状态信息，它被表示为一个以NULL字符结尾的字符串；如果执行出错，则将返回NULL值。这个字符串的格式并不固定，下面是它在最近一些MySQL版本里的样子：

```
Uptime: 1189474  Threads: 4  Questions: 331869  Slow queries: 50  Opens: 1424
Flush tables: 1  Open tables: 64  Queries per second avg: 0.279
```

这个字符串的各组成部分的含义如下所示：

- Uptime——MySQL服务器自启动以来的无故障运行时间，以秒为计算单位。
- Threads——MySQL服务器里现在正在运行的线程的个数。
- Questions——MySQL服务器已经执行过的数据库查询命令的个数。
- Slow queries——慢查询的个数；如果MySQL服务器用来处理某个查询命令的时间超出了服务器变量long\_query\_time所设定的时间，这个查询就将被认为是“慢查询”。
- Opens——MySQL服务器曾经打开过的数据表的个数。
- Flush tables——MySQL服务器曾经执行过的FLUSH、REFRESH和RELOAD语句的总个数。
- Open tables——MySQL服务器现在打开的数据表的个数。
- Queries per second——Questions和Uptime的比值。

mysql\_stat()函数所返回的信息与mysqladmin status命令的输出报告有着同样的内容，这并不是巧合——mysqladmin程序正是通过调用这个函数的办法来获取有关信息的。

#### • unsigned long

```
mysql_thread_id ( MYSQL *conn );
```

返回值：与给定连接相关联的MySQL服务器线程的ID号。可以把这个ID号当做线程标识符传递给mysql\_kill()函数。

在调用了mysql\_thread\_id()函数之后，必须立刻让它返回的ID号派上用场。如果你提前获得了这个值并把它保存了起来，等你稍后再想使用这个值的时候，它有可能已经发生了改

变。这种情况多发生在你的连接是在掉线后又重新建立起来的场合（比如说，使用 `mysql_ping()` 函数）；此时，MySQL 服务器将给新连接重新分配一个线程标识符。

### F.3.6 管理类函数

这一小节里的函数将使你能够对MySQL服务器的操作情况进行控制。

- int

**mysql\_shutdown ( MYSQL \*conn );**

通知MySQL服务器关机。必须具备SHUTDOWN权限才能调用这个函数。

如果调用成功，`mysql_shutdown()`将返回零；如果执行出错，则将返回一个非零值。

### F.3.7 线程类函数

这一小节里的函数是供你编写多线程客户程序用的。

- void

**mysql\_thread\_end ( void );**

释放由`mysql_thread_init ()`函数调用初始化出来的、与线程处理机制有关的各个变量。为避免出现内存泄漏现象，必须明确地调用这个函数来结束你创建的每一个线程。

`mysql_thread_end ()`函数最早出现于MySQL 4.0.0版本。

- my-bool

**mysql\_thread\_init ( void );**

对与线程处理机制有关的各个变量进行初始化。只要你创建的线程将会调用MySQL函数，就必须首先调用这个函数。此外，在结束这类线程之前，还必须调用`mysql_thread_end ()`函数去释放各有关变量。

`mysql_thread_init ()`函数最早出现于MySQL 4.0.0版本。

- unsigned int

**mysql\_thread\_safe ( void );**

如果你使用的MySQL C客户程序开发库支持多线程机制，这个函数的返回值将是1；否则，它的返回值就将是0。这个函数的返回值反映了你在配置MySQL时是否使用了`--enable-thread-safe-client`选项。

`mysql_thread_safe ()`函数最早出现于MySQL 3.23.14版本。

### F.3.8 与嵌入式MySQL服务器libmysqld进行通信的函数

这一小节里介绍的函数是用来与嵌入式MySQL服务器libmysqld进行通信用的。在开发一个需要用到嵌入式MySQL服务器的应用软件的过程中，必须使用`-lmysqld`而不是`-lmysqlclient`库链接选项来链接有关的程序才能生成内嵌有MySQL服务器功能的可执行映像来。此外，要想在系统上把嵌入式MySQL服务器链接到你的应用软件里，可能还需要用到其他一些库链接标志，下面这条命令可以把这些链接标志给查出来：



```
% mysql_config --libmysqld-libs
```

对应于-lmysqlclient选项的普通MySQL C客户程序开发库包含用来与嵌入式MySQL服务器libmysqld进行通信的函数的定义，但那些函数只有调用模型而没有具体的实现代码。也就是说，如果你在程序里调用了这些函数，那么，对于同样的源代码，如果使用-lmysqlclient选项去链接，就能生成一个可独立运行的客户程序；如果使用-lmysqld选项去链接，就能生成一个内嵌有MySQL服务器功能的客户程序。

- void

```
mysql_server_end ( void );
```

关闭嵌入式MySQL服务器。在使用完嵌入式MySQL服务器之后，别忘了调用这个函数去关闭它。

- int

```
mysql_server_init ( int argc,
                    char **argv,
                    char **groups );
```

对嵌入式MySQL服务器进行初始化。这个函数必须在你调用任何一个mysql\_xxx()函数之前最先被调用。

这个函数的argc和argv参数与我们传递给C语言程序中的main()函数的标准参数是一样的：argc是输入参数的个数（如果没有输入参数，argc将等于0；否则，argc就是你准备传递给嵌入式MySQL服务器的输入参数的个数）；argv是一个以NULL字符结尾的字符串向量，其内容是各有关参数的值（注意：argv[0]将被忽略）。

groups参数也是一个以NULL字符结尾的字符串向量，其内容是你打算让嵌入式MySQL服务器到各有关选项文件里去读取的选项组的名单；这个向量的最后一个元素必须是NULL。如果groups参数本身就是NULL，嵌入式MySQL服务器将默认地去读取各有关选项文件里的[server]和[embedded]选项组里的选项。注意：在groups向量里给出的选项组名称不需要用“[”和“]”字符括起来。

### F.3.9 调试类函数

下面这些函数允许你在客户端或服务器端生成调试信息。要想使用这些函数，你的MySQL软件必须编译有调试功能的支持机制才行——也就是说，你在配置MySQL发行版本时必须使用过--with-debug选项或--with-debug=full选项。使用--with-debug=full选项将导致产生更多的调试信息并会激活safemalloc机制，safemalloc是一个带高级检查功能的内存分配函数库。

- void

```
mysql_debug ( const char *debug_str );
```

使用字符串debug\_str进行一次DEBUG\_PUSH操作。这个字符串的格式可以在MySQL Reference Manual（MySQL参考手册）里查到。

要想使用mysql\_debug()函数，MySQL C客户程序开发库必须编译有调试支持功能。

- int

**mysql\_dump\_debug\_info** ( MYSQL \*conn );

让MySQL服务器把调试信息写到日志里去。在MySQL 4.0.2之前的版本里, 必须具备PROCESS权限才能调用这个函数; 在MySQL 4.0.2及以后的版本里, 必须具备SUPER权限。如果调用成功, mysql\_dump\_debug\_info ()函数的返回值将是0; 否则, 它将返回一个非零值。

### F.3.10 正逐渐被淘汰的函数

MySQL C客户程序开发库里还有一些正逐渐被淘汰的函数, 因为它们已经有了更好的替代去完成同样的工作。如今, 在过去需要调用这些函数才能完成的工作有很多都可以通过使用mysql\_query()函数发出一个等价的数据库查询命令的方法去完成。比如说, mysql\_create\_db ("db\_name")现在可以用如下所示的函数调用去完成:

```
mysql_query (conn, "CREATE DATABASE db_name" )
```

随着时间的推移, MySQL能够识别和处理的SQL语句越来越多, 因此而遭到淘汰的MySQL C API函数也就越来越多。比如说, 当SQL语句FLUSH PRIVILEGES被增加到MySQL里的时候, mysql\_reload()函数的使用价值就降低了; 而随着FLUSH语句的功能的进一步扩展, mysql\_refresh()函数的使用价值也降低了。在后面的讨论内容里, 我们将在介绍各有关函数的时候注明它开始遭到淘汰的MySQL版本以及人们现在用来完成同样工作的替代办法。不过, 如果你现在使用的MySQL C客户程序开发库的版本低于我们所列举的MySQL版本, 你就还得使用那些老版本的函数才行——如果真是这样的话, 建议你尽快对它进行升级。一般说来, 把MySQL C客户程序开发库升级到一个更新的版本是很安全的, 因为它既能配合老版本的MySQL服务器工作, 也能配合新版本的MySQL服务器工作。

有些函数 (比如mysql\_connect()、mysql\_eof()、mysql\_escape\_string()等等) 之所以会遭到淘汰, 是因为它们的工作现在可以用功能更多、反馈信息也更多的其他函数来完成。

从长远考虑, 在编写MySQL客户程序的时候, 希望大家尽量避免使用这一小节里列举的各个函数, 因为它们当中的一部分或者是全部很可能会彻底“消失”。

- MYSQL \*

```
mysql_connect ( MYSQL *conn,
                const char *host_name,
                const char *user_name,
                const char *password );
```

这个函数是mysql\_real\_connect ()函数的前身。它现在被实现为对mysql\_real\_connect ()函数的一个调用。

这个函数是从MySQL 3.22.0版本开始遭到淘汰的。

- int

**mysql\_create\_db** ( MYSQL \*conn, const char \*db\_name );

用db\_name参数给定的名字创建一个数据库。这项工作现在可以通过使用mysql\_query()函

数发出一个CREATE DATABASE语句的办法来完成。

如果调用成功，mysql\_create\_db()函数的返回值将是0；如果执行出错，它将返回一个非零值。

这个函数是从MySQL 3.21.15版本开始遭到淘汰的。

- int

```
mysql_drop_db ( MYSQL *conn,
                 const char *db_name );
```

根据db\_name参数给定的名字丢弃数据库。这项工作现在可以通过使用mysql\_query()函数发出一个DROP DATABASE语句的办法来完成。

如果调用成功，mysql\_drop\_db()函数的返回值将是0；如果执行出错，它将返回一个非零值。

这个函数是从MySQL 3.21.15版本开始遭到淘汰的。

- my\_bool

```
mysql_eof ( MYSQL_RES *res_set );
```

如果已经到达结果集的末尾，则返回一个非零值；如果执行出错，则返回零。mysql\_eof()函数是MySQL C客户程序开发库为配合mysql\_use\_result()和mysql\_fetch\_row()函数的使用而提供的。我们知道，mysql\_use\_result()只对结果集进行初始化，它并不会把结果集里的数据行取回到客户主机上来；而这种场合里的mysql\_fetch\_row()调用每次只能从MySQL服务器那里取回一个数据行，如果它的返回值是NULL，那么既有可能是已经到达了结果集的末尾，也可能是在执行过程中发生了错误。因此，我们要靠mysql\_eof()来区分这两种情况。

现在，人们可以通过mysql\_errno()和mysql\_error()函数来获得同样的效果。当然，这两个出错处理函数返回的信息也更多了——它们不仅能告诉你是否发生了错误，还能告诉你错误发生的原因。

这个函数是从MySQL 3.21.17版本开始遭到淘汰的。

- unsigned long

```
mysql_escape_string ( char *to_str,
                     const char *from_str,
                     unsigned long from_len );
```

对包含有特殊字符的字符串进行编码，使它能够在一条SQL语句里。mysql\_escape\_string()函数在功能上与mysql\_real\_escape\_string()函数相似，但它没有conn参数，只能对当前连接上的字符串进行转义处理；在编码时没有把当前字符集的因素考虑在内。详细情况请参见前面介绍mysql\_real\_escape\_string()函数时的讨论。

这个函数是从MySQL 3.23.14版本开始遭到淘汰的，现在的推荐做法是使用mysql\_real\_escape\_string()函数来代替之。

在MySQL 3.23.6之前的版本里，这个函数的from\_len参数是unsigned int类型。

• int

**mysql\_kill** ( MYSQL \*conn, unsigned long thread\_id );

终止执行thread\_id参数所给定的MySQL服务器线程。如果调用成功，它的返回值将是零；如果执行出错，它将返回一个非零值。你永远能终止你自己的线程。如果想终止别人的线程，那你还必须具备相应的PROCESS权限（在MySQL 4.0.2之前的版本里）或SUPER权限（在MySQL 4.0.2及以后的版本里）才行。

这个函数的工作现在可以通过使用mysql\_query()函数发出一个KILL语句的办法完成。

这个函数是从MySQL 3.22.9版本开始遭到淘汰的。

• MYSQL\_RES \*

**mysql\_list\_dbs** ( MYSQL \*conn, const char \*wild );

返回一个结果集，其内容是一份与conn参数值相关联的MySQL服务器上的数据库名单。如果执行出错，则将返回NULL值。这份名单包括且仅包括那些名字与wild参数所给定的SQL匹配模式（即以“%”和“\_”字符作为通配符的匹配模式）相匹配的数据库；如果wild参数的值是NULL，与conn参数值相关联的MySQL服务器上的所有数据库就都将出现在这份名单里。调用mysql\_list\_dbs()函数而生成的结果集要由你去负责释放——可以用mysql\_free\_result()函数来完成这项工作。

通过调用mysql\_list\_dbs()函数而得到的数据库名单现在可以通过先使用mysql\_query()函数发出一个SHOW DATABASES语句、再对结果集进行相应处理的办法来获得。

这个函数是从MySQL 3.22.0版本开始遭到淘汰的。

• MYSQL\_RES \*

**mysql\_list\_fields** ( MYSQL \*conn,  
                    const char \*tbl\_name,  
                    const char \*wild );

返回一个结果集，其内容是一份给定数据表的数据列名单。如果执行出错，则将返回NULL值。这份名单包括且仅包括那些名字与wild参数所给定的SQL匹配模式（即以“%”和“\_”字符作为通配符的匹配模式）相匹配的数据列；如果wild参数的值是NULL，给定数据表的所有数据列就都将出现在这份名单里。调用mysql\_list\_fields()函数而生成的结果集要由你去负责释放——可以用mysql\_free\_result()函数来完成这项工作。

通过调用mysql\_list\_fields()函数而得到的数据列名单现在可以通过先使用mysql\_query()函数发出一个SHOW COLUMNS语句、再对结果集进行相应处理的办法来获得。

这个函数是从MySQL 3.22.0版本开始遭到淘汰的。

• MYSQL\_RES \*

**mysql\_list\_processes** ( MYSQL \*conn );

返回一个结果集，其内容是一份当时正在MySQL服务器里运行着的进程名单。如果执行出错，则将返回NULL值。如果你具备PROCESS权限，这份名单将包括所有的MySQL服务器进程；如果你不具备PROCESS权限，这份名单将只包括你自己的进程。调用mysql\_list\_processes()函数而生成的结果集要由你去负责释放——可以用mysql\_free\_result()

函数来完成这项工作。

通过调用mysql\_list\_processes()函数而得到的进程名单现在可以通过先使用mysql\_query()函数发出一个SHOW PROCESSLIST语句、再对结果集进行相应处理的办法来获得。

这个函数是从MySQL 3.22.0版本开始遭到淘汰的。

- **MYSQL\_RES \***

**mysql\_list\_tables** ( MYSQL \*conn, char \*wild );

返回一个结果集，其内容是一份当前数据库里的数据表名单。如果执行出错，则将返回NULL值。这份名单包括且仅包括那些名字与wild参数所给定的SQL匹配模式（即以“%”和“\_”字符作为通配符的匹配模式）相匹配的数据表；如果wild参数的值是NULL，当前数据库里的所有数据表就都将出现在这份名单里。调用mysql\_list\_tables()函数而生成的结果集要由你去负责释放——可以用mysql\_free\_result()函数来完成这项工作。

通过调用mysql\_list\_tables()函数而得到的数据表名单现在可以通过先使用mysql\_query()函数发出一个SHOW TABLES语句、再对结果集进行相应处理的办法来获得。

这个函数是从MySQL 3.22.0版本开始遭到淘汰的。

- **int**

**mysql\_refresh** ( MYSQL \*conn, unsigned int options );

这个函数的使用效果与SQL语句FLUSH和RESET相同，但它允许你同时给出多个选项，从而达到只用一个函数调用就刷新（指把缓存区里的内容写到磁盘——清空，或者把磁盘上的原始数据重新读入缓存区——重置）多个MySQL缓存区的效果。如果调用成功，mysql\_refresh()函数的返回值将是0；如果执行出错，它将返回一个非零值。

这个函数的options参数由下列选项中的一个或者多个组成。必须具备RELOAD权限才能进行这些操作。

- **REFRESH\_GRANT**

重新加载各权限表的内容，相当于发出一条FLUSH PRIVILEGES语句。

- **REFRESH\_HOSTS**

刷新主机缓存区，相当于发出一条FLUSH HOSTS语句。

- **REFRESH\_LOG**

用先关闭再打开的办法来刷新日志文件，适用于MySQL服务器已经打开的任何日志。它相当于发出一条FLUSH LOGS语句。

- **REFRESH\_MASTER**

让镜像机制中的主服务器删除那些名字出现在二进制日志索引文件里的二进制日志并把二进制日志索引文件的长度截短为零，相当于发出一条RESET MASTER语句。

这个选项是从MySQL 3.23.19版本开始遭到淘汰的。

- **REFRESH\_SLAVE**

让镜像机制中的从服务器忘记它的主日志里的位置，相当于发出一条RESET SLAVE语句。

这个选项是从MySQL 3.23.19版本开始遭到淘汰的。

- **REFRESH\_STATUS**

把各状态变量重新设置为它们各自的初始值，相当于发出一条RESET STATUS语句。



这个选项是从MySQL 3.22.11版本开始遭到淘汰的。

- **REFRESH\_TABLES**

关闭所有被打开的数据表，相当于发出一条FLUSH TABLES语句。

- **REFRESH\_THREADS**

刷新线程缓存区。

这个选项是从MySQL 3.23.16版本开始遭到淘汰的。

上面这些选项标志都是位值，所以可以用二进制位操作符“|”或者“+”把它们组合在一起使用以得到叠加的效果。比如说，下面这两个表达式就是等价的：

```
REFRESH_LOG | REFRESH_TABLES  
REFRESH_LOG + REFRESH_TABLES
```

mysql\_refresh()函数遭到淘汰的原因是你现在可以通过发出FLUSH或RESET语句的办法来完成同样的动作。本书的附录D对这两条SQL语句进行了详细的介绍。

- **int**

**mysql\_reload ( MYSQL \*conn );**

让服务器重新加载权限表。这项工作现在可以用以mysql\_query()函数发出一个FLUSH PRIVILEGES查询的办法来完成。必须具备RELOAD权限才能调用mysql\_reload()函数。

如果调用成功，mysql\_reload()函数的返回值将是0；如果执行出错，它将返回一个非零值。

这个函数是从MySQL 3.21.9版本开始遭到淘汰的。



## 附录G Perl DBI API指南

本附录对用来开发MySQL应用程序的Perl DBI应用程序设计接口进行介绍。利用这个API所提供的方法和属性，Perl脚本能与数据库服务器进行通信并对数据库进行访问。本附录还将对MySQL数据库驱动模块DBD::mysql向DBI提供的MySQL扩展功能进行介绍。本附录没有讨论较早出现的Mysqlperl接口，因为它已经过时了。

有些DBI模块的方法和属性没有在本附录里介绍，它们或者不适用于MySQL，或者是一些比较新的试验性方法，还有待于完善或者最终被放弃。有些与MySQL有关的DBD方法也没有收录在本附录里，因为它们已经过时了。如果你想了解这些新、旧功能，请自行查阅DBI文档或者MySQL DBD文档，可以用下面这些命令来查看它们：

```
% perldoc DBI
% perldoc DBI::FAQ
% perldoc DBD::mysql
```

这里并不想把这篇指南写成一本科教书，所以其中收录的都是一些演示Perl DBI API使用方法的代码片段。在本书的第7章里，大家可以看到一些完整的脚本以及编写它们的指导意见。

### G.1 编写Perl DBI脚本

如果你想在Perl脚本里使用DBI模块，就必须把下面这行代码加到你的脚本代码里去：

```
use DBI;
```

但你不必用一条use指令来导入某个具体的DBD级别的模块，因为DBI会在你连接数据库服务器时自动启用正确的模块。

在正常情况下，DBI脚本需要使用connect()方法来打开一个与MySQL服务器的连接，再使用disconnect()方法来关闭这个连接；只有在这个连接处于打开状态期间，你才可以发出各种数据库查询命令。用来执行查询命令的方法有好几种，应该根据数据库查询语句的类型来选用它们。非SELECT查询通常是用do()方法来直接执行的。SELECT查询通常要分成以下几个步骤来执行：先把查询命令传递到prepare()方法，再调用execute()方法去执行它，最后用一个数据行取回方法（比如fetchrow\_array()或fetchrow\_hashref()）在一个循环语句里以一次一个数据行的方式来取回并处理它的查询结果。

当你通过DBI脚本来发出数据库查询命令的时候，每个查询命令字符串只能包含一条SQL语句，并且不允许在末尾加上分号(;)或“\g”结束符；结束符“;”和“\g”都是mysql客户程序使用的表示法，DBI脚本所发出的数据库查询命令不需要它们来表示SQL语句的结束。

### G.2 DBI方法

在本书的附录F和附录H里，我们在介绍各有关C语言函数和PHP函数时给出的是一个完整

的函数框架，那些函数的返回值类型和输入参数类型都有明确的定义。但在这个附录里，在介绍DBI方法的时候采用了一种不同的格式。我们将用一些变量来代表各有关方法的输入参数和返回值，而它们的类型将通过变量名的第一个字符来隐含地表明：'\$'代表一个标量值，'@'代表一个数组， '%'代表一个散列值（即关联数组）。此外，如果变量名的前面还有一个'\字符，则表示它是一个指向变量的引用指针而不是变量本身。如果变量名还有一个ref后缀，则表示变量的值是一个引用指针。

表G-1列出了本附录里比较常见的一些变量和它们的基本含义。

表G-1 常用的Perl DBI变量名

变 量 名	含 义
\$drh	一个句柄，指向一个驱动模块对象
\$dbh	一个句柄，指向一个数据库对象
\$sth	一个句柄，指向一条语句（数据库查询命令）对象
\$fh	一个句柄，指向一个已被打开的文件
\$h	一个“通用”句柄，其具体含义需要根据上下文确定
\$rc	返回代码，来自那些会返回真值或假值的操作
\$rv	返回值，来自那些会返回整数的操作
\$rows	返回值，来自那些会返回数据行计数值的操作
\$str	返回值，来自那些会返回字符串的操作
@ary	一个数组，它的用途需要根据上下文确定
@row_ary	一个数组，其中存放着从一条查询命令的结果集里取回的一个数据行的内容

此外，很多DBI方法都接受散列参数%attr，这个参数给出的属性值将影响到这个方法的具体工作情况。在用到这个散列参数的时候，应该把它的引用指针传递给有关方法，其具体做法有两种：

- 先设置好散列值%attr的键值，然后通过再调用有关的方法并把这个散列值的引用指针用做那个方法的一个输入参数，如下所示：

```
my %attr = (AttrName1 => value1, AttrName2 => value2 );
$ret_val = $h->method (... , \%attr );
```

- 直接在方法调用中使用一个匿名散列值，如下所示：

```
$ret_val = $h->method (... , {AttrName1 => value1, AttrName2 => value2 } );
```

DBI方法或DBI函数的调用方式将由调用序列表明。“DBI->”表明是一个DBI类方法，“DBI::”表明是一个DBI函数，“\$DBI::”表明是一个DBI变量。对于那些需要通过句柄来调用的方法，句柄的名称将表明方法的作用范围。“\$dbh->”表明是一个数据库句柄方法，“\$sth->”表明是一个语句句柄方法，“\$h->”表明方法可以用各种不同的句柄来调用。对于那些允许省略的可选信息，我们将把它们放在一对方括号（[]）里。请看下面这个调用序列示例：

```
@row_ary = $dbh->selectrow_array ($statement, [\%attr [, @bina_values ] ] );
```

上面这个调用序列里的“\$dbh->”表明selectrow\_array()方法是作为一个数据库句柄方法而被调用的。这个方法的输入参数是\$statement（一个标量值）、%attr（一个散列值，因为它前面

有一个“\”字符，所以我们知道它必须被传递为一个引用指针)和@bind\_values(一个数组)，而且第2和第3个输入参数是可选的。它的返回值@row\_ary是一个数组，其中存放着这个方法所返回的数据行的内容。

在对各有关方法进行介绍的时候，我们会给出它在执行出错时的返回值，但这个值只有在我们事先禁用了RaiseError属性的情况下才会被返回。如果RaiseError属性处于激活状态，执行出错的方法将不会返回值而是会抛出一个异常并导致脚本自动退出执行。

在下面的内容里，“SELECT查询”代表着能够返回一些数据行的各种数据库查询命令，比如SELECT、DESCRIBE、EXPLAIN或SHOW等等。

### G.2.1 DBI类方法

在这一小节里，各有关方法的%attr参数可以用来设定方法的属性。(如果某个属性参数被省略或者取值为undef，则表示“不具备该属性”。)对与MySQL有关的DBI方法来说，最重要的属性是PrintError、RaiseError和AutoCommit。被传递给connect()或connect\_cached()方法的属性将成为这两个方法所返回的结果数据库句柄的一部分。比如说，下面这行代码在创建数据库句柄\$dbh的时候激活了RaiseError属性，这就激活了脚本的出错自动退出机制：如果脚本在调用与数据库句柄\$dbh相关联的任何一个方法时发生了一个DBI错误，脚本就将自动退出执行：

```
$dbh = DBI->connect ($data_source, $user_name, $password, {RaiseError => 1} );
```

我们将在本附录后面内容里的第G.4节对PrintError、RaiseError和AutoCommit属性做详细的讨论。

- @ary = DBI->available\_drivers ([ \$quiet ] );

返回一份可用DBI模块的名单。可选参数\$quiet的默认值是0，用来在发现多个驱动模块有同样的名字时报告一条警告消息。如果你不想看到这样的警告消息，把\$quiet的值设置为1即可。

- \$dbh = DBI->connect (\$data\_source,  
                      \$user\_name,  
                      \$password  
                      [, \%attr ] );

connect()方法负责建立与MySQL服务器的连接。如果调用成功，它将返回一个数据库句柄，如果调用失败，则返回undef。在成功地建立起一个与MySQL服务器的连接之后，如果你想关闭它，就需要使用connect()所返回的数据库句柄来调用disconnect()方法。如下所示：

```
$dbh = DBI->connect ("DBI:mysql:sampdb:cobra.snake.net",  
                      "sampadm", "secret", \%attr)  
      or die "Could not connect\n";  
$dbh->disconnect ();
```

数据源(\$data\_source)允许以几种不同的格式给出。数据源必须是以DBI:mysql:开头，其中，DBI这几个字符的大小写可以随意，mysql这几个字符却必须是小写。注意，因为

第二个冒号（这个冒号不允许省略）后面的全部内容都将由具体的数据库驱动模块负责解释，所以本附录后面内容里介绍的语法不一定适用于DBD::mysql以外的其他数据库驱动模块。

在第二个冒号的后面，还可以给出一个数据库名和一个主机名，它们将与DBI:mysql:共同构成数据源字符串的开头部分。如下所示：

```
$data_source = "DBI:mysql:db_name";
$data_source = "DBI:mysql:db_name:host_name";
```

数据源字符串中的数据库名允许以`db_name`或`database = db_name`形式给出，主机名允许以`host_name`或`host = host_name`形式给出。

还可以在数据源字符串开头部分的后面紧接着以`attribute = value`的格式给出一些选项。这些选项必须以分号加以分隔（即每个选项的前面都必须有一个分号）。如下所示：

```
DBI:mysql:sampdb:localhost;mysql_socket=/tmp/mysql.sock;mysql_compression=1
```

MySQL驱动模块能够识别并支持使用以下几个选项：

- `host = host_name`

脚本将去连接的主机。对于TCP/IP连接，还可以通过`host_name:port_num`格式或直接使用`port`属性来指定一个端口号。

在UNIX系统上，对本地主机localhost的连接将默认地使用UNIX套接字来建立。（这需要你通过`mysql_socket`属性来给出一个套接字名。）如果想使用TCP/IP来连接本地主机，就需要以`host = 127.0.0.1`的形式来加以指定。

在基于Windows NT的系统上，如果你试图建立一个到主机“.”的连接，DBI将默认地使用命名管道去连接本地服务器（这需要你通过`mysql_socket`属性来指定一个命名管道）；但如果DBI无法通过这个命名管道连接上主机“.”，它就会使用TCP/IP去建立这个连接。

- `port = port_num`

脚本将去连接的端口号。脚本在建立非TCP/IP连接时（比如你在UNIX系统上试图连接本地主机localhost时）将忽略这个选项。

- `mysql_client_found_rows = val`

在默认的情况下，当你发出一条UPDATE查询时，MySQL服务器将返回一个计数值，但这个计数值只能报告这条UPDATE查询影响（修改）了多少个数据行，而不能告诉你这条UPDATE查询匹配到了多少个数据行。如果你想知道UPDATE查询匹配到了多少个数据行，就需要把这个`mysql_client_found_rows`选项设置为1。把这个选项设置为0表示你不想改变MySQL服务器的默认行为。

`mysql_client_found_rows`选项最早出现于DBD::mysql 1.2208版本。

- `mysql_compression = 1`

这个选项负责激活客户与MySQL服务器之间的压缩通信。

`mysql_compression`选项最早出现于DBD::mysql 1.1920版本，并要求MySQL的版本号不低于3.22.3。

- `mysql_connect_timeout = seconds`



以秒为计量单位的连接倒计时等待时间。如果未能在这段时间内成功地建立起与MySQL服务器的连接，connect()就会返回undef以表明失败。

mysql\_connect\_timeout选项最早出现于DBD::mysql 1.2207版本。

- `mysql_local_infile = val`

可以在MySQL 3.23.49及以后的版本里通过这个选项来激活/禁用LOAD DATA语句的LOCAL能力。如果LOCAL机制在MySQL客户程序开发库里的默认设置是处于禁用状态，把这个选项设置为1将激活之（但前提是MySQL服务器本身没有被配置成不允许使用LOCAL机制的情况）。如果LOCAL机制在MySQL客户程序开发库里的默认设置是处于激活状态，把这个选项设置为0将禁用之。

mysql\_local\_infile选项最早出现于DBD::mysql 2.1020版本。

- `mysql_read_default_file = file_name`

在默认的情况下，DBI脚本是不会到任何一个MySQL选项文件里去读取连接参数的；如果想让脚本到某个选项文件里去读取连接参数，可以用mysql\_read\_default\_file选项来给出这个文件的文件名。这个文件名最好是以完整路径名的形式给出。（否则，DBI脚本就会把它解释为一个相对于当前目录的路径，你在其他目录里执行这个脚本的时候就会遇到麻烦。）把连接参数硬编码在脚本代码里并不是值得提倡的做法。如果你知道自己编写的某个脚本将会有很多人使用，就应该让这个脚本到那些人各自的选项文件里读取连接参数。在UNIX系统上，可以把mysql\_read\_default\_file选项里的文件名设置为\$ENV{HOME}/.my.cnf。这样，谁正在执行这个脚本，脚本就会到谁的主目录里去寻找选项文件.my.cnf并使用那个文件里的连接参数去连接MySQL服务器。

但是，在Windows系统上使用mysql\_read\_default\_file选项时会遇到这样一个问题：Windows的文件路径名通常以一个驱动器盘符和一个冒号(:)开始，可DBI却会把这个冒号解释为数据源字符串里的分隔符。虽说有一个能绕开这一限制的办法，但这个办法却显得有点笨拙：1) 先把路径切换到选项文件所在驱动器的根目录，这样就可以用一个不带驱动器盘符的相对路径来给出选项文件的路径名；2) 把数据源字符串里的mysql\_read\_default\_file选项设置为选项文件的文件名，但不要写出该驱动器的盘符和紧跟在盘符后面的冒号；3) 如果还想在连接上MySQL服务器之后再回到当前目录里来，就必须在调用connect()之前先把当前目录的路径名保存起来。等连接操作完成后，再通过chdir()调用重新回去。

mysql\_read\_default\_file选项最早出现于DBD::mysql 1.2106版本，并要求MySQL的版本号不低于3.22.10。

- `mysql_read_default_group = group_name`

指定一个选项文件组，DBI脚本将依次到几个标准的选项文件里去读取你在指定的选项文件组里给出的连接参数。如果只设定了mysql\_read\_default\_group而没有设定mysql\_read\_default\_file选项，DBI脚本将到一组标准的选项文件里去寻找连接参数；如果同时设定了这两个选项，DBI脚本将只读取mysql\_read\_default\_file选项所指定的选项文件。mysql\_read\_default\_file选项只能让脚本去读取指定的选项文件，如果还想让它去读取全局选

项文件（比如UNIX系统上的/etc/my.cnf文件或者Windows系统上的C:\my.cnf文件）里的连接参数，它就不能胜任了。如果你想让自己的脚本依次读取所有标准选项文件里的连接参数，就应该使用mysql\_read\_default\_group选项。这个选项将把各标准选项文件里的[client]选项组和你指定的选项组里的连接参数都读取出来。比如说，mysql\_read\_default\_group = dbi将使脚本去读取各选项文件里的[dbi]和[client]选项组。如果只想读取[client]选项组，就应该把这个选项写成mysql\_read\_default\_group = client。

对选项文件的格式的介绍请参见附录E。

mysql\_read\_default\_group选项最早出现于DBD::mysql 1.2106版本，并要求MySQL的版本号不低于3.22.10。

- **mysql\_socket = socket\_name**

在UNIX系统上，需要通过这个选项来指定DBI脚本在连接本地主机localhost时使用的UNIX套接字。在Windows系统上，需要用这个选项来指定命名管道。脚本在建立TCP/IP连接时（比如你在UNIX系统上试图连接本地主机localhost以外的其他主机时）将忽略这个选项。

mysql\_socket选项要求MySQL的版本号不低于3.21.15。

- **mysql\_ssl = 1**

mysql\_ssl\_ca\_file = file\_name

mysql\_ssl\_ca\_path = dir\_name

mysql\_ssl\_cipher = str

mysql\_ssl\_client\_cert = file\_name

mysql\_ssl\_client\_key = file\_name

这些选项只有在DBI脚本准备使用SSL去建立一条与服务器之间的安全化连接时才需要给出。把mysql\_ssl选项设置为1将激活SSL机制；再使用其他几个选项来设定这个连接的其他特性。这些选项与C API中的mysql\_ssl\_set()函数所使用的相应参数含义相同，详细情况请参见本书附录F中的有关条目。如果激活了mysql\_ssl选项，那你至少还需要再对mysql\_ssl\_ca\_file、mysql\_ssl\_client\_cert、mysql\_ssl\_client\_key三个选项进行设定。

这些选项最早出现于DBD::mysql 2.1013版本，并要求使用MySQL 4或更高的版本。

如果没有在connect()调用的输入参数里明确地给出某些连接参数，DBI将检查以下几个环境变量以确定将要使用哪些连接参数去连接MySQL服务器：

- 如果数据源没有定义或者是一个空字符串，connect()将使用环境变量DBI\_DSN的值作为数据源。
- 如果没有在数据源字符串里给出数据库驱动模块的名字，connect()将使用环境变量DBI\_DRIVER所指定的数据库驱动模块。
- 如果connect()方法的user\_name和password参数没有定义（注意，不包括它们是空字符串的情况），将使用环境变量DBI\_USER和DBI\_PASS的值作为用户名和口令。不过使用环境变量DBI\_PASS的值作为口令是不安全的，在多用户系统上应该尽量避免这样做，因为其他用户往往能够通过一些系统监控命令查看到环境变量的明文内容。

如果DBI在检查过上述信息源之后仍有一些连接参数不能确定，它就会使用一些默认值来完成connect()调用。比如说，如果是主机名无法确定，DBI就将连接本地主机localhost；如果是用户名无法确定，DBI就将以你的登录名（如果使用的是UNIX系统）或ODBC（如果使用的是Windows系统）为默认的用户名。但如果是口令无法确定，DBI就没有默认值了——它将不发送口令。

- `$dbh = DBI->connect_cached ( $data_source,  
                                  $user_name,  
                                  $password  
                                  [, \%attr ] );`

这个方法与connect()只有一个区别：DBI将把connect\_cached()所返回的数据库句柄缓存起来。如果脚本在前一个connect\_cached()所返回的数据库句柄依然有效期间又使用完全一样的连接参数发出了一个connect\_cached()调用，DBI将简单地返回那个被缓存起来的数据库句柄而不会再另外去创建一个新的连接。如果那个被缓存起来的数据库句柄已经失效，DBI就将建立一个新的连接并缓存和返回这个新数据库句柄。

- `@ary = DBI->data_sources ( $driver [, \%attr ] );`

返回一份现在能够通过数据库驱动模块\$driver进行访问的数据库名单。与MySQL相对应的数据库驱动模块\$driver就是mysql（注意要小写）。如果\$driver是undef或者是一个空字符串，DBI将检查环境变量DBI\_DRIVER以确定数据库驱动模块的名字。

很多DBI驱动模块上的data\_sources()调用都只能返回一个空名单或者一个不完整的名单。这个问题在MySQL上表现得尤其突出：这个函数只有在数据库驱动模块DBI::mysql能够以完全匿名（无用户名、无口令）的方式连接到本地主机上的某个MySQL服务器的情况下才能返回一份有用的数据库名单。可又有谁会把自己的MySQL服务器配置得这么不安全呢？

- `$drh = DBI->install_driver ( $driver_name );`

激活一个DBD级的驱动模块并返回一个驱动模块句柄，若无法找到指定的驱动模块，脚本将在显示一条出错信息后退出执行。与MySQL相对应的数据库驱动模块\$driver就是mysql（注意要小写）。你一般没有必要去亲自调用这个方法，因为DBI会在你发出connect()调用的时候自动激活正确的数据库驱动模块。不过，如果你想通过func()方法去执行一些管理性操作（见本附录后面内容里的第G.2.5节）的话，install\_driver()还是有用的。

## G.2.2 数据库句柄方法

这一小节里的方法都需要使用一个数据库句柄来调用，在使用这些方法之前，应该先通过上一节介绍的connect()或connect\_cached()调用去获得一个这样的句柄。

在这一小节里，各有关方法的%attr参数可以用来设定方法的属性。（如果某个属性参数被省略或者取值为undef，则表示“不具备该属性”。）对与MySQL有关的DBI方法来说，最重要的属性是PrintError和RaiseError。比如说，如果RaiseError属性当前处于禁用状态，下面这行代码将在对语句\$statement进行处理期间激活它，如果在此期间发生了一个DBI错误，脚本就将自

动退出执行：

```
$rows = $dbh->do ($statement, {RaiseError => 1} );
```

我们将在本附录后面内容里的第G.4节对PrintError和RaiseError属性做详细的讨论。

- `$src = $dbh->begin_work ();`

这个调用将把数据库句柄属性AutoCommit临时改变为禁用状态并关闭自动提交模式，这将使我们能够开始执行一个由多条语句组成的事务。在下一个commit()或rollback()调用发生之前，AutoCommit将一直保持在禁用状态；在下一个commit()或rollback()调用发生之后，AutoCommit将自动恢复为激活状态。使用begin\_work()方法与以手动方式把AutoCommit属性设置为禁用状态是不同的；在后一种场合，AutoCommit不会在commit()或rollback()调用发出之后自动恢复为激活状态，必须由你以手动方式重新激活它才行。

如果AutoCommit被成功地设置为禁用状态，begin\_work()方法将返回真；如果它已经处于禁用状态，则返回假。如果MySQL不支持事务处理，把AutoCommit设置为禁用状态的尝试将导致一个致命错误。

begin\_work()方法最早出现于DBI 1.20版本。事务支持要求DBD::mysql模块的版本号不低于1.2216且MySQL的版本号不低于3.23.17。

- `$src = $dbh->commit ();`

提交当前事务——前提是MySQL支持事务处理且AutoCommit处于禁用状态。若AutoCommit处于激活状态，发出commit()调用将没有任何效果并会导致一条警告信息。

事务支持要求DBD::mysql模块的版本号不低于1.2216且MySQL的版本号不低于3.23.17。

- `$src = $dbh->disconnect ();`

断开与数据库句柄\$dbh相关联的连接。如果某个连接在脚本退出执行时仍保持在连通状态，DBI将自动断开这个连接并给出一条警告信息。

DBI没有对在事务处理的过程中发出的disconnect()调用的行为做出定义。正确的做法是：如果事务尚未完成，就应该在发出disconnect()调用之前先发出一个commit()或rollback()调用。

- `$rows = $dbh->do ( $statement  
                    [, \%attr  
                    [, @bind_values ] ] );`

准备并执行\$statement参数给出的数据库查询命令。若调用成功，这个方法将返回一个告诉你这次查询影响（修改）了多少个数据行的计数值；若因为某种原因无法确定受其影响的数据行的个数，则返回-1；若执行出错，将返回undef。如果受其影响的数据行个数是零，这个方法的返回值将是字符串"0E0"。"0E0"在数值上下文里将被求值为数值0，在布尔上下文里将被视为真。

do()方法的主要用途是执行那些不会返回数据行的SQL语句，例如DELETE、INSERT、REPLACE或UPDATE等。用do()方法去执行一条SELECT语句是不恰当的——因为无法得到一个语句句柄，所以你将无法取回任何一个数据行。

一般来说，在调用do()方法的时候通常用不着传递什么属性，所以它的%attr参数几乎总是



被设置为undef。@bind\_value参数对应着一个数组，它代表着一组将被绑定到占位符上的值，占位符是一些我们在构造查询命令字符串时安排在其中的“?”字符。

如果查询命令里没有占位符，do()调用里的%attr参数和@bind\_value值就都可以省略，如下所示：

```
$rows = $dbh->do ("UPDATE member SET expiration = NOW() WHERE member_id = 39" );
```

但如果查询命令字符串里真的包含有占位符，%attr参数就不允许省略，而且@bind\_value数组里的元素个数也必须与查询命令字符串里的占位符个数相等。在下面这个do()调用里，因为查询命令字符串里有两个占位符，所以我们要把它的属性参数%attr设置为undef，然后再给出两个用来替换占位符的绑定值：

```
$rows = $dbh->do ("UPDATE member SET expiration = ? WHERE member_id = ?",
                 undef,
                 "2005-11-30", 39);
```

- `$src = $dbh->ping ();`

在连接上MySQL服务器之后，如果脚本长时间没有进行数据库操作，服务器就会在一段倒计时时间结束后自动断开这个连接。ping()调用将重新建立这条连接。如果连接仍处于连通状态或者成功地重新连接上了MySQL服务器，ping()调用将返回真；否则，将返回假。

- `$sth = $dbh->prepare ( $statement [, \%attr ] );`

对\$statement参数给出的查询命令进行预处理以便在今后执行之。如果调用成功，prepare()将返回一个语从句柄；如果执行出错，则返回undef。如果prepare()调用成功，脚本就可以用它返回的语从句柄去发出execute()调用以执行数据库查询命令了。

- `$sth = $dbh->prepare_cached ( $statement  
 [, \%attr  
 [, $allow_active ] ] );`

这个方法与prepare()只有一个区别：DBI将把prepare\_cached()所返回的语从句柄缓存起来。如果脚本在前一个prepare\_cached()所返回的语从句柄依然有效期间又使用同样的\$statement和%attr参数发出了一个prepare\_cached()调用，DBI将简单地返回那个被缓存起来的语从句柄而不会再另外去创建一个新的句柄。在返回那个被缓存起来的语从句柄之前，DBI会先发出一个finish()调用并给出一条警告信息。给\$allow\_active参数传递一个真值将抑制（即不显示）这条警告信息。

- `$str = $dbh->quote ( $value [, $data_type ] );`

有些字符在SQL里有着特殊的含义和作用，如果不加处理地让这些字符出现在数据库查询命令的数据值里，在执行这条查询命令时就会出现语法错误。为避免出现这种情况，就需要对查询命令里的字符串数据值进行处理。quote()就是用来完成这一任务的，它将给参数\$value给出的字符串加上适当的引号并对其中的SQL特殊字符进行转义。比如说，字符串“I'm happy”在经过quote()处理之后将变成“'I'm happy'”。如果\$value参数的取值是undef，quote()将返回单词NULL。需要提醒大家注意的是，从quote()返回的字符串都已经被加上必要的引号了，所以在往查询命令字符串里插入这些字符串的时候就不应该再画蛇添足地



给它们加引号了。

不要使用quote()方法对将来用来替换占位符的数据值进行预处理——DBI会自动地给这些数据值加上必要的引号，否则，这个数据值将会有多余的引号。

MySQL能够根据上下文把查询命令里的字符串数据值转换为必要的数据类型，所以quote()方法的\$data\_type参数通常用不着给出。不过，如果你想亲自指定某项数据的类型，或者想提醒自己不要忘记某项数据应该是某种特定类型的话，就需要给出一个\$data\_type参数。比如说，可以用DBI::SQL\_INTEGER来表明\$value参数代表的是一个整数。

- `$rc = $dbh->rollback ();`

回滚当前事务——前提是MySQL支持事务处理且AutoCommit处于禁用状态。若AutoCommit处于激活状态，发出rollback()调用将没有任何效果并会产生一条警告信息。

事务支持要求DBD::mysql模块的版本号不低于1.2216且MySQL的版本号不低于3.23.17。

- `$ary_ref = $dbh->selectall_arrayref ( $statement  
  [, \%attr  
  [, @bind_values ] ] );`

执行\$statement参数所给出的数据库查询命令。从效果上讲，这个方法的调用效果相当于连续发出相应的prepare()、execute()和fetchall\_arrayref()等三个调用。如果\$statement参数已经是一个语句句柄（也就是说，在发出selectall\_arrayref()调用之前已经用prepare()方法对该语句进行过预处理），selectall\_arrayref()将省略（即不进行）相应的预处理步骤。%attr和@bind\_values参数的含义和用法与它们在do()方法中的情况相同。

如果调用成功，selectall\_arrayref()将返回一个数组引用指针，这个数组里的元素本身又是一些数组引用指针，它们所指向的数组分别对应着结果集里的一个数据行。如果结果集里没有数据行（即\$statement语句没能在数据库里检索到符合条件的数据行），selectall\_arrayref()返回的引用指针将指向一个空数组。

如果\$statement语句尚未执行就发生了错误，selectall\_arrayref()将返回undef。如果在\$statement语句的执行过程中发生了错误，selectall\_arrayref()将把出错之前已经检索到的全部数据行返回给脚本。如果你想知道非undef返回值代表的是成功还是失败，可以发出一个\$dbh->err()调用或者查看\$DBI::err属性。

- `$hash_ref = $dbh->selectall_hashref ( $statement, $key_col  
  [, \%attr  
  [, @bind_values ] ] );`

执行\$statement参数所给出的数据库查询命令。从效果上讲，这个方法的调用效果相当于连续发出相应的prepare()、execute()和fetchall\_hashref()等三个调用。如果\$statement参数已经是一个语句句柄（也就是说，在发出selectall\_hashref()调用之前已经用prepare()方法对该语句进行过预处理），selectall\_hashref()将省略（即不进行）相应的预处理步骤。%attr和@bind\_values参数的含义和用法与它们在do()方法中的情况相同。

如果调用成功，selectall\_hashref()将返回一个散列值引用指针，这个散列值里的各个元素的键字是\$key\_col参数指定的数据列的取值，它或者是数据库查询命令所选取的某个数据

列的名称，或者是一个数据列编号（从1开始）；而这个散列值里的各个元素的键值则又是一个散列值引用指针，它们所指向的散列值分别对应着结果集里的一个数据行并以 \$statement 语句所选取的各数据列的名称作为各散列元素的键字。为避免出现因散列值里的键字同名而导致的数据行丢失现象，\$key\_col 参数所指定的数据列里的取值应该各不相同。如果结果集里没有数据行（即 \$statement 语句没能在数据库里检索到符合条件的数据行），selectall\_hashref() 返回的引用指针将指向一个空散列表。

如果 \$statement 语句尚未执行就发生了错误，selectall\_hashref() 将返回 undef。如果在 \$statement 语句的执行过程中发生了错误，selectall\_hashref() 将把出错之前已经检索到的全部数据行返回给脚本。如果你想知道非 undef 返回值代表的是成功还是失败，可以发出一个 \$dbh->err() 调用或者查看 \$DBI::err 属性。

selectall\_hashref() 方法最早出现于 DBI 1.20 版本。（其实它在 DBI 1.15 版本里就已经出现了，但那时的行为与现在不一样。）

- `$ary_ref = $dbh->selectcol_arrayref ( $statement  
  [, \%attr  
  [, @bind_values ] ] );`

执行 \$statement 参数所给出的数据库查询命令。从效果上讲，这个方法的调用效果相当于连续发出相应的 prepare()、execute() 和一个数据行取回调用。如果 \$statement 参数已经是一个语句句柄（也就是说，在发出 selectcol\_arrayref() 调用之前已经用 prepare() 方法对该语句进行过预处理），selectcol\_arrayref() 将省略（即不进行）相应的预处理步骤。%attr 和 @bind\_values 参数的含义和用法与它们在 do() 方法中的情况相同。

如果调用成功，selectcol\_arrayref() 将返回一个数组引用指针，这个数组里的各个元素分别对应着结果集中的各个数据行的第一个数据列。如果结果集里没有数据行（即 \$statement 语句没能在数据库里检索到符合条件的数据行），selectcol\_arrayref() 返回的引用指针将指向一个空数组。

如果 \$statement 语句尚未执行就发生了错误，selectcol\_arrayref() 将返回 undef。如果在 \$statement 语句的执行过程中发生了错误，selectcol\_arrayref() 将把出错之前已经检索到的全部数据行返回给脚本。如果你想知道非 undef 返回值代表的是成功还是失败，可以发出一个 \$dbh->err() 调用或者查看 \$DBI::err 属性。

selectcol\_arrayref() 方法最早出现于 DBI 1.09 版本。

- `@row_ary = $dbh->selectrow_array ( $statement  
  [, \%attr  
  [, @bind_values ] ] );`

执行 \$statement 参数所给出的数据库查询命令。从效果上讲，这个方法的调用效果相当于连续发出相应的 prepare()、execute() 和 fetchrow\_array() 等三个调用。如果 \$statement 参数已经是一个语句句柄（也就是说，在发出 selectrow\_array() 调用之前已经用 prepare() 方法对该语句进行过预处理），selectrow\_array() 将省略（即不进行）相应的预处理步骤。%attr 和 @bind\_values 参数的含义和用法与它们在 do() 方法中的情况相同。

在列表上下文里，如果调用成功，`selectrow_array()`将返回一个数组，这个数组对应着结果集里的第一个数据行；如果结果集里没有数据行（即`$statement`语句没能在数据库里检索到符合条件的数据行）或执行出错，`selectrow_array()`将返回一个空数组。在标量上下文里，如果调用成功，`selectrow_array()`将返回与结果集里的第一个数据行相对应的那个数组里的某一个元素（不过，DBI并没有对这个返回值到底是哪一个元素做出具体的规定；另请参见第G.2.3节里`fetchrow_array()`条目下的有关说明）；如果结果集里没有数据行（即`$statement`语句没能在数据库里检索到符合条件的数据行）或`selectrow_array()`本身执行出错，则将返回`undef`。

在列表上下文里，如果你想知道这个方法的调用结果是“没有返回数据行”还是“执行出错”，可以发出一个`$dbh->err()`调用或者查看`$DBI::err`属性——零值代表“没有返回数据行”。在标量上下文里，如果执行没有出错，`undef`返回值的含义将难以确定：既有可能是数据列的取值就是`NULL`，也有可能是“没有返回数据行”。

- `$ary_ref = $dbh->selectrow_arrayref ( $statement  
  [, \%attr  
  [, @bind_values ] ] );`

执行`$statement`参数所给出的数据库查询命令。从效果上讲，这个方法相当于你连续发出了相应的`prepare()`、`execute()`和`fetchrow_arrayref()`等三个调用。如果`$statement`参数已经是一个语句句柄（也就是说，在发出`selectrow_arrayref()`调用之前已经用`prepare()`方法对该语句进行过预处理），`selectrow_arrayref()`将省略（即不进行）相应的预处理步骤。`%attr`和`@bind_values`参数的含义和用法与它们在`do()`方法中的情况相同。

如果调用成功，`selectrow_arrayref()`将返回一个数组引用指针，这个数组对应着结果集里的第一个数据行；如果结果集里没有数据行（即`$statement`语句没能在数据库里检索到符合条件的数据行），`selectrow_arrayref()`返回的引用指针将指向一个空数组；如果执行出错，则将返回`undef`。

`selectrow_arrayref()`方法最早出现于DBI 1.15版本。

- `$hash_ref = $dbh->selectrow_hashref ( $statement  
  [, \%attr  
  [, @bind_values ] ] );`

执行`$statement`参数所给出的数据库查询命令。从效果上讲，这个方法相当于你连续发出了相应的`prepare()`、`execute()`和`fetchrow_hashref()`等三个调用。如果`$statement`参数已经是一个语句句柄（也就是说，在发出`selectrow_hashref()`调用之前已经用`prepare()`方法对该语句进行过预处理），`selectrow_hashref()`将省略（即不进行）相应的预处理步骤。`%attr`和`@bind_values`参数的含义和用法与它们在`do()`方法中的情况相同。

如果调用成功，`selectrow_hashref()`将返回一个散列值，这个散列值对应着结果集里的第一个数据行并以`$statement`语句所选取的各数据列的名称作为各散列元素的键字。如果结果集里没有数据行（即`$statement`语句没能在数据库里检索到符合条件的数据行），

`selectrow_hashref()`返回的引用指针将指向一个空散列表；如果执行出错，则将返回`undef`。  
`selectrow_hashref()`方法最早出现于DBI 1.16版本。

#### 其他数据库句柄方法

DBI在自己的新版本里增加了一些用来获取数据库和数据表元数据的数据库句柄方法。但它们当中有很多现在仍处于试验阶段，并且大都没有在MySQL上实现。这些方法包括`column_info()`、`foreign_key_info()`、`get_info()`、`primary_key()`、`primary_key_info()`、`table_info()`、`tables()`、`type_info()`和`type_info_all()`等。它们的详细情况可以在DBI文档中查到：

% perldoc DBI

### G.2.3 语句句柄方法

这一小节里的方法都需要使用一个语句句柄来调用，在使用这些方法之前，应该先通过上一节介绍的`prepare()`或`prepare_cached()`调用去获得一个这样的句柄。

- `$rc = $sth->bind_col ( $col_num, \ $var );`

把SELECT查询所选取的某一个数据列（由参数`$col_num`指定）绑定到一个由参数`$var`指定的Perl变量（注意，这个变量必须以一个引用指针的方式给出）上。脚本每取回一个数据行，`$var`参数所指定的Perl变量就会自动更新为指定数据列的新值。`$col_num`的取值范围是从1到这个查询所选取的数据列的个数；如果`$col_num`参数指定的数据列编号超出了这个范围，`bind_col()`调用将返回一个假值。

`bind_col()`调用应该发生在`execute()`调用之后、开始取回数据行之前。

- `$rc = $sth->bind_columns ( \ $var1, \ $var2, ... );`

把SELECT查询所选取的各个数据列依次绑定到由参数`$var1`、`$var2`等指定的一组Perl变量（注意，这些变量必须以引用指针的方式给出）上。参见对`bind_col()`方法的描述。与`bind_col()`方法的情况类似，`bind_columns()`调用也应该发生在`execute()`调用之后、开始取回数据行之前。

如果`$var1`、`$var2`等参数的个数与查询命令所选取的数据列个数不匹配，`bind_columns()`调用将返回一个假值。

- `$rv = $sth->bind_param ( $n, $value [, \%attr ] );`

`$rv = $sth->bind_param ( $n, $value [, $bind_type ] );`

把一个数据值绑定到查询命令字符串里的一个占位符上，使这个值随着查询命令一起发送到MySQL服务器上去。占位符在查询命令字符串里被表示为问号“?”。`bind_param()`调用应该发生在`prepare()`调用之后、`execute()`调用之前。

数据值`$value`将被绑定到编号为`$n`的占位符上，`$n`的取值范围是从1到占位符的个数。如果需要绑定的数据值是NULL，`$value`就应该是`undef`。

在默认的情况下，DBI会把被绑定数据视为一个VARCHAR值。因为MySQL能够根据上下文把查询命令里的字符串数据值转换为必要的数据类型，所以这个方法里的`%attr`和`$bind_type`参数通常用不着给出。不过，如果你想亲自指定被绑定数据的类型，或者想提



醒自己不要忘记被绑定数据应该是某种特定类型的话，就需要给出它们了。比如说，下面两条\$bind\_param()调用都可以用来把被绑定数据限定为一个整数：

```
$rv = $sth->bind_param ($n, $value, { TYPE => DBI::SQL_INTEGER });
$rv = $sth->bind_param ($n, $value, DBI::SQL_INTEGER);
```

- \$rows = \$sth->dump\_results ( [ \$maxlen [, \$line\_sep [, \$field\_sep [, \$fh ] ] ] ] );

取回语句句柄\$sth上的所有数据行，调用DBI工具函数DBI::neat\_list()对它们进行排版并把它们打印到参数\$fh指定的文件里去。这个方法的返回值是受其影响的数据行的个数。

\$maxlen、\$line\_sep、\$field\_sep、\$fh等参数的默认值分别是：35、“\n”、“和STDOUT。

- \$rv = \$sth->execute ( [ @bind\_values ] );

把语句句柄\$sth所指定的数据库查询命令发送到MySQL服务器去执行。对于SELECT类语句，如果调用成功，execute()将返回一个真值；如果执行出错，则返回undef。对于非SELECT类语句，如果调用成功，execute()将返回一个告诉你这条语句影响（修改）了多少个数据行的计数值；若因为某种原因无法确定受其影响的数据行的个数，则返回-1；若执行出错，将返回undef。如果受其影响的数据行个数是零，这个方法的返回值将是字符串“0E0”，“0E0”在数值上下文里将被求值为数值0，在布尔上下文里将被视为真。

@bind\_values参数的含义和用法与它们在do()方法中的情况相同。

- \$ary\_ref = \$sth->fetch ();

fetch()是fetchrow\_arrayref()方法的别名。

- \$ary\_ref = \$sth->fetchall\_arrayref ( [ \$slice\_array\_ref ] );

\$ary\_ref = \$sth->fetchall\_arrayref ( [ \$slice\_hash\_ref ] );

取回语句句柄\$sth上的所有数据行并返回一个数组引用指针，这个数组里的元素也是一些引用指针，它们依次指向结果集里的每一个数据行。如果结果集里没有数据行，fetchall\_arrayref()返回的引用指针将指向一个空数组。否则，数组\$ary\_ref里的各个元素（一些引用指针）将它们依次指向结果集里的每一个数据行。这些数据行引用指针的含义要根据你传递给这个方法的输入参数来确定。如果fetchall\_arrayref()调用没有使用输入参数或者输入参数是一些数组下标值，这些数据行引用指针就将分别指向一个由数据列值构成的数组。数组下标值将从0开始计算——因为它们给出的是某个Perl数组的下标，负值表示从数据行的尾端倒数计算。比如说，如果你想把各个数据行的第一个和最后一个数据列提取出来，就需要使用下面这样的代码：

```
$ary_ref = $sth->fetchall_arrayref ( [0, -1] );
```

如果fetchall\_arrayref()调用的输入参数是一些散列下标值，这些数据行引用指针就将分别指向一个由数据列值构成的散列值，这些散列值的键字就是你在数据库查询命令里给出的那些数据列的名称。散列下标值的使用方法是：以数据列的名字作为键字，以1作为键值，如下所示：

```
$ary_ref = $sth->fetchall_arrayref ( {id => 1, name => 1} );
```

如果想把所有数据列全都取到一个散列值里去，就需要使用一个空散列值的引用指针作



为fetchall\_arrayref()调用的输入参数，如下所示：

```
$ary_ref = $sth->fetchall_arrayref ( { } );
```

即使执行出错，fetchall\_arrayref()也能把出错之前已经检索到的数据行全部返回给脚本。如果你想知道这个方法在调用过程中是否发生过错误，可以发出一个\$dbh->err()调用或者查看\$DBI::err属性。

- `$hash_ref = $sth->fetchall_hashref ( $key_col );`

取回结果集。如果调用成功，这个方法将返回一个散列值引用指针，这个散列值里的各个元素的键字是\$key\_col参数指定的数据列的取值，它或者是数据库查询命令所选取的某个数据列的名称，或者是一个数据列编号（从1开始）；而这个散列值里的各个元素的键值则又是一个散列值引用指针，它们所指向的散列值分别对应着结果集里的一个数据行并以数据库查询命令所选取的数据列的名字作为各散列元素的键字。为避免出现因散列值里的键字同名而导致的数据行丢失现象，\$key\_col参数所指定的数据列里的取值应该各不相同。如果结果集里没有数据行（即前一个execute()调用没能在数据库里检索到符合条件的数据行），fetchall\_hashref()返回的引用指针将指向一个空散列表。

如果是因\$key\_col参数不合法而导致的执行出错，fetchall\_hashref()将返回undef；否则，即使执行出错，它也能把出错之前已经检索到的数据行全部返回给脚本。如果你想知道非undef返回值代表的是成功还是失败，可以发出一个\$dbh->err()调用或者查看\$DBI::err属性。

- `@ary = $sth->fetchrow_array ();`

在列表上下文里，如果调用成功，这个方法将把结果集里的下一个数据行取回到数组@ary里；如果已经到达结果集里的最后一个数据行或执行出错，则将返回一个空数组。在标量上下文里，如果调用成功，这个方法将取回结果集里的下一个数据行的某一个元素；如果已经到达结果集里的最后一个数据行或执行出错，则将返回undef。1.29版本之前的DBI文档说fetchrow\_array()在标量上下文里将返回第一个数据列的值，但大家在编写DBI脚本的时候千万不要依赖这一行为，因为新版本的DBI没有对fetchrow\_array()方法在标量上下文里将返回哪个数据列的行为做出明确的规定；DBI 1.29之前版本的说法只有在结果集里只包含有一个数据行的时候才是准确的。

如果你想知道这个方法的undef返回值的含义是“正常到达结果集的末尾”还是“发生了错误”，可以发出一个\$dbh->err()调用或者查看\$DBI::err属性——零值代表“正常到达结果集的末尾”。在标量上下文里，如果执行没有出错，undef返回值的含义将难以确定：既有可能是数据列的取值就是NULL，也有可能是“没有返回数据行”。

- `$ary_ref = $sth->fetchrow_arrayref ();`

如果调用成功，这个方法将把结果集里的下一个数据行提取到引用指针\$ary\_ref所指向的一个数组里；如果已经到达结果集里的最后一个数据行或执行出错，则将返回undef。如果你想知道这个方法的undef返回值的含义是“正常到达结果集的末尾”还是“发生了一个错误”，可以发出一个\$dbh->err()调用或者查看\$DBI::err属性——零值代表“正常到达结果集的末尾”。

- `$hash_ref = $sth->fetchrow_hashref ( [ $name ] );`

如果调用成功，这个方法将把结果集里的下一个数据行提取到引用指针`$hash_ref`所指向的一个散列值里；如果已经到达结果集里的最后一个数据行或执行出错，则将返回`undef`。散列元素的键字是数据列的名字，键值则是数据列里的数据值。

`$name`参数（如果给出的话）用来控制散列键字的字母大小写情况。它的默认值是“NAME”（意思是以数据库查询命令中给出的数据列名称作为键字）。把`$name`参数设置为“NAME\_lc”或“NAME\_uc”将把构成散列键字的字符全都强制转换为小写或大写字母。（`FetchHashKeyName`属性也是控制散列键字大小写情况的办法之一，详细情况请参见第G.4节。）

如果你想知道这个方法的`undef`返回值的含义是“正常到达结果集的末尾”还是“发生了一个错误”，可以发出一个`$dbh->err()`调用或者查看`$DBI::err`属性——零值代表“正常到达结果集的末尾”。

- `$rc = $sth->finish ();`

释放与语句句柄`$sth`相关联的全部资源。我们通常用不着亲自去发出一个对这个方法的调用，因为那些用来取回数据行的DBI方法会在到达结果集里的最后一个数据行时隐含地自动发出一个`finish()`调用。但如果你只取回了结果集的一部分，就应该明确地发出一个`finish()`调用以通知DBI说你已经完成了语句句柄上的数据行取回工作，可以释放与之有关的资源了。

发出`finish()`调用之后，有关的语句属性都将失效。又因为那些用来取回数据行的DBI方法会在检测到结果集末尾的时候隐含地自动发出一个`finish()`调用，所以你最好是在调用完`execute()`方法之后立刻去访问各有关属性。

- `$rv = $sth->rows ();`

返回一个数据行计数值，即有多少个数据行受到了与语句句柄`$sth`相关联的数据库查询命令的影响；如果执行出错，则将返回-1。`rows()`方法多被用来检查UPDATE或DELETE等不会返回数据行的语句的执行情况。对SELECT类语句最好不要使用`rows()`方法；应该一边从结果集里提取数据行，一边统计数据行的个数。

#### G.2.4 通用句柄方法

这一小节里的方法不需要通过特定类型的句柄来调用，可以使用驱动模块句柄、数据库句柄或语句句柄来调用它们。

- `$rv = $h->err ();`

返回最近一次调用的DBI操作的数值出错代码。就MySQL而言，就是由MySQL服务器返回的错误代号。如果这个返回值是0或`undef`，就表明调用执行过程没有出错。

- `$str = $h->errstr ();`

返回最近一次调用的DBI操作的字符串出错信息。就MySQL而言，就是由MySQL服务器返回的出错信息。如果这个返回值是一个空字符串或`undef`，就表明调用执行过程没有出错。

- `DBI->trace ( $trace_level [, $trace_filename ] );`

`$h->trace ( $trace_level [, $trace_filename ] );`

设置一个跟踪调试级别。DBI提供的跟踪调试机制能够让我们掌握DBI操作的执行情况。跟踪级别从0（关闭）到9（信息量最大）共分10档。可以对脚本里的所有DBI操作都进行跟踪调试（办法是把`trace()`作为一个DBI类方法来调用），也可以只对某个特定的句柄进行跟踪调试，如下所示：

```
DBI->trace (2);           # 激活对整个脚本的跟踪调试
$sth->trace (2);          # 只激活对语句句柄$sth的跟踪调试
```

如果想对自己运行的所有DBI脚本都进行跟踪调试，最简便的办法是设置`DBI_TRACE`环境变量。

在默认的情况下，调试信息将被发送到系统的标准出错设备`STDERR`去。但如果你在调用`trace()`方法的时候通过`$trace_filename`参数给出了一个文件名，就可以把调试信息重定向到这个文件里去。跟踪调试机制的输出信息将被追加在这个文件末尾而不会覆盖掉它原有的内容，建议大家定期清理这个文件。

注意，来自所有被跟踪句柄的调试信息将都写入同一个文件。如果你在发出`trace()`调用时还给出了一个文件名，那么所有的调试信息将都写入这个文件；如果你没有给出这个参数，那么所有的调试信息将都送往`STDERR`设备。

- `DBI->trace_msg ( $str [, $min_level ] );`

`$h->trace_msg ( $str [, $min_level ] );`

如果把`trace_msg()`当做一个类方法来调用（`DBI->trace_msg()`），那么，只要跟踪调试机制已经在DBI全局层次上被激活，它就会把调试信息`$str`写到调试输出设备去。如果把`trace_msg()`当做一个句柄方法来调用（`$h->trace_msg()`），那么，只有在你对这个句柄做过`trace()`调用或者在DBI全局层次激活了跟踪调试机制的情况下，它才会把调试信息`$str`写到调试输出设备去。

如果还给出了一个`$min_level`参数，那么调试信息`$str`将只有在跟踪调试级别至少是`$min_level`参数所指定的级别时才会被写到调试输出设备去。

### G.2.5 用来执行MySQL数据库管理操作的方法

我们将在这一小节向大家介绍如何使用DBI提供的`func()`方法来直接完成一些数据库驱动模块级的操作。注意，`func()`方法与数据库中的存储过程毫不相干。（有些数据库引擎允许用户把自行开发的、用来访问数据库内容的可执行代码保存在数据库服务器里供今后使用，这种代码就是所谓的`stored procedure`。这个术语目前还没有统一的中文译法，有些译者把它翻译为“程序模块”、“库内过程”，请大家在阅读其他书刊的时候注意这一点。——译者注）DBI目前还不支持从客户端去调用MySQL服务器里的存储过程。

- `$src = $drh->func ( "createdb",  
                    $db_name, $host_name,  
                    $user_name, $password,  
                    "admin" );`

```

$rc = $drh->func ( "dropdb",
                  $db_name, $host_name,
                  $user_name, $password,
                  "admin" );
$rc = $drh->func ( "reload",
                  $host_name, $user_name,
                  $password, "admin" );
$rc = $drh->func ( "shutdown",
                  $host_name, $user_name,
                  $password, "admin" );
$rc = $dbh->func ( "createdb", $db_name, "admin" );
$rc = $dbh->func ( "dropdb", $db_name, "admin" );
$rc = $dbh->func ( "reload", "admin" );
$rc = $dbh->func ( "shutdown", "admin" );

```

func()方法既可以通过数据库驱动模块句柄来调用，也可以通过数据库句柄来调用。因为数据库驱动模块句柄并不与某个已经打开的连接相关联，所以如果想以这种方式来调用func()方法的话，就必须给出相应的主机名、用户名、口令等参数才行——func()方法需要使用这些参数去连接数据库服务器。如果你是通过数据库句柄来调用func()方法的，就不需要给出这些参数了。如有必要，可以通过下面这个办法来获得一个数据库驱动模块句柄：

```
$dbh = DBI->install_driver ("mysql");      # 注意：“mysql”必须小写
```

func()方法能够识别和进行的操作动作有以下几种：

- **createdb** 以\$db\_name参数给出的名字创建一个数据库。必须有这个数据库上的CREATE权限才能执行这个操作。
- **dropdb** 丢弃（即删除）\$db\_name参数指定的数据库。必须有这个数据库上的DROP权限才能执行这个操作。

请三思而后行；在执行了数据库丢弃操作之后，数据库里的数据就找不回来了。

- **reload** 重新加载MySQL数据库中的各有关权限表。这个操作只有在你是以DELETE、INSERT或UPDATE等语句直接修改了权限表的内容时才有必要使用；如果你是以GRANT、REVOKE等语句去完成权限授予和收回工作的，就不需要发出这个调用。必须有这个数据库上的RELOAD权限才能执行这个操作。
  - **shutdown** 关闭服务器。必须有这个数据库上的SHUTDOWN权限才能执行这个操作。
- 注意，在上述操作中，只有shutdown是唯一能够通过正常的DBI查询处理机制来进行的func()动作。其他动作最好是以CREATE DATABASE、DROP DATABASE或FLUSH PRIVILEGES等SQL语句而不是以发出func()调用的办法去完成。

### G.3 DBI工具函数

DBI还为我们准备了一些辅助性的工具例程，可以用它们来完成数据值的测试和打印输出工

作。这些函数必须以DBI::func\_name()而不是DBI->func\_name()的形式去调用。

- @bool = DBI::looks\_like\_number ( @ary );

对数组@ary中的各个元素看起来是否像数字进行测试。这个方法的返回值是一个布尔数组@bool，该数组里的每一个成员对应着@ary数组中的一个元素。如果@ary数组中的某个元素看起来像数字，@bool数组中与之对应的元素就是一个真值；如果不像，返回一个假值；如果某个@ary元素未定义或者为空，相应的@bool元素就将返回undef。

- \$str = DBI::neat ( \$value [, \$maxlen ] );

对\$value参数给出的数据值进行排版并返回结果字符串。如果\$value是一个字符串，则给它加上引号；如果它是一个数值，则不加引号（注意：用引号引起来的数值将被视为字符串）；如果它是一个未定义值，则返回undef；如果是非打印字符，则返回“.”。我们来看一个例子：

```
for my $val ( "a", "3", 3, undef, "\x01\x02" )
{
    print DBI::neat ( $val ) . "\n";
}
```

如果执行上面这个循环，将得到如下所示的输出：

```
'a'
'3'
3
undef
'...'
```

\$maxlen参数控制着结果的最大长度；如果结果的长度超过了\$maxlen，DBI就将把它截短为\$maxlen-4个字符并在其后加上“...”以表明这种情况。如果参数\$maxlen等于0或者没有给出，它的默认值将等于\$DBI::neat\_maxlen的当前值，\$DBI::neat\_maxlen本身的默认值是400。

不要用neat()函数来构造查询命令字符串，给将被插入查询命令字符串的数据值添加引号或者对之进行转义处理的工作应该使用占位符或者通过quote()调用去完成。

- \$str = DBI::neat\_list ( \@ary  
                            [, \$maxlen  
                            [, \$sep ] ] );

调用neat()函数对数组@ary中的各个元素依次进行处理，再用分隔符\$sep把它们合并起来并返回为一个字符串。

\$maxlen参数将被传递到neat()函数，所以它只对@ary数组中的各个元素有影响，最终的结果字符串不受这个参数的影响。

如果\$sep参数没有被给出，它的默认值将是", "（一个逗号加一个空格）。

## G.4 DBI属性

DBI还提供有多种层次的属性信息。大多数属性或者被关联到数据库句柄、或者被关联到语



句句柄，但不允许同时被关联到这两种句柄上。但有些属性（比如PrintError或RaiseError）却既可以被关联到数据库句柄也可以被关联到语句句柄。一般说来，每个句柄都有它自己的属性，但那些用来存放出错信息的属性（比如err和errstr）却是“动态”的，它们不属于某个特定的句柄，而是被“动态地”关联到你最近一次使用的那个句柄上去。

被传递给connect()或connect\_cached()方法的属性将成为这两个方法所返回的结果数据库句柄的一部分。

#### G.4.1 数据库句柄属性

本小节里的DBI属性都是与数据库句柄相关联的。

- **\$dbh->{ AutoCommit };**

把这个属性设置为真或假就可以激活或者禁用自动提交模式，它的默认设置是真。把AutoCommit属性设置为假将使我们能够开始执行一个由多条语句组成的事务：如果这些语句全都执行成功，就可以发出一个commit()调用来提交这次事务（使有关语句对数据库的改动生效）；只要这些语句里有一条没有执行成功，就应该发出一个rollback()调用把数据库回滚为这次事务开始之前的样子。另请参见本附录前面内容里对数据库句柄方法begin\_work()的描述。

如果MySQL不支持事务处理，把AutoCommit属性设置为禁用状态的尝试将导致一个致命错误。事务支持要求DBD::mysql模块的版本号不低于1.2216且MySQL的版本号不低于3.23.17。

- **\$dbh->{ Statement };**

这个属性的值是DBI脚本最近一次通过给定的数据库句柄\$dbh传递给prepare()方法的SQL查询命令字符串。

#### G.4.2 通用句柄属性

下面这些属性既可以被关联到某个具体的句柄（\$h->{Attribute\_name}），也可以出现在各有关方法的属性参数%attr里（{Attribute\_name => value}）以影响该方法的操作行为。

- **\$h->{ ChopBlanks };**

这个属性决定着那些用来从结果集里取回数据行的DBI方法是否会去掉字符数据列里的数据值尾部的空格：如果把它设置为真，则去掉空格；如果把它设置为假，则不去掉空格。绝大多数数据库驱动模块里的ChopBlanks属性的默认值都是假。这个属性对CHAR或VARCHAR数据列没有影响，因为MySQL服务器本身就会去掉这两种数据列里的数据值尾部的多余空格。不过，对于BLOB或TEXT数据列，把ChopBlanks属性设置为真将导致数据列里的数据值尾部的空格被去掉。

- **\$h->{ FetchHashKeyName };**

fetchrow\_hashref()以及其他几个需要调用fetchrow\_hashref()来完成操作的DBI方法会把从结果集里取回的数据行返回为一些散列值，FetchHashKeyName属性控制着这些散列值里由各有关数据列名称充当的键字的字母大小写情况。这个属性的默认设置值是"NAME"（意思是以数据库查询命令中给出的数据列名称作为键字），它设置为"NAME\_lc"或

"NAME\_uc"将把那些由数据列名称充当的散列键字全都强制转换为小写或大写字母。这个属性只适用于数据库句柄和数据库驱动模块句柄。

FetchHashKeyName属性最早出现于DBI 1.19版本。

- **\$h->{ HandleError };**

这个属性是用来进行出错处理的。可以把它设置为一个你自己编写的出错处理子例程的引用指针（即该例程的入口地址），当执行发生错误的时候，DBI将在调用RaiseError或PrintError等常规出错处理例程之前先调用你的出错处理例程：如果这个例程的返回值是真，DBI将跳过（即不调用）RaiseError或PrintError等常规出错处理例程；如果这个例程的返回值是假，DBI将在这个例程的调用结束之后进一步调用RaiseError或PrintError等常规例程进行出错处理。（当然，你编写的这个例程完全可以终止脚本的执行而不是返回。）DBI将向出错处理例程传递三个参数——出错消息的文本、出错时脚本正在使用的DBI句柄、执行出错的方法所返回的第一个值。

HandleError属性最早出现于DBI 1.21版本。

- **\$h->{ PrintError };**

如果这个属性被设置为真，与DBI有关的执行出错就会产生一条警告消息被打印出来。这个属性的默认设置值是假。这个属性对执行出错的DBI方法的返回值并没有影响，它只决定它们是否需要在返回之前打印一条出错消息。

- **\$h->{ RaiseError };**

如果这个属性被设置为真，与DBI有关的执行出错就会导致脚本自动结束运行。这个属性的默认设置值是假。

- **\$h->{ ShowErrorStatement };**

如果这个属性被设置为真，DBI就会把执行出错的数据库查询命令的文本追加在出错消息的末尾一起打印出来。这个属性的默认设置值是假。这个属性的效力只限于各语句句柄以及prepare()与execute()方法。

ShowErrorStatement属性最早出现于DBI 1.19版本。

- **\$h->{ TraceLevel };**

设置或查知指定句柄上的跟踪调试级别。这个属性相当于trace()方法的一个替代品。

TraceLevel属性最早出现于DBI 1.21版本。

### G.4.3 MySQL特有的数据库句柄属性

本小节里的属性都是DBI MySQL驱动模块DBI::mysql所特有的。正如大家将会看到的那样，这些属性分别对应着MySQL C API里的某个函数。详细情况请参见本书的附录F。

- **\$rv = \$dbh->{ mysql\_errno };**

返回最近一次发生的DBI错误的出错代码。这个属性与C API中的mysql\_errno()函数功能相当。

- **\$str = \$dbh->{ mysql\_error };**

返回最近一次发生的DBI错误的出错字符串。这个属性与C API中的mysql\_error()函数功能相当。

- `$str = $dbh->{ mysql_hostinfo };`  
返回对给定连接进行描述的字符串。这个属性与C API中的`mysql_get_host_info()`函数功能相当。
  - `$str = $dbh->{ mysql_info };`  
有些数据库查询命令会影响到多个数据行，这个属性所返回的就是有关这些查询命令的描述信息。这个属性与C API中的`mysql_info()`函数功能相当。
  - `$rv = $dbh->{ mysql_insertid };`  
返回与\$dbh相关联的数据库连接上最近一次生成的AUTO\_INCREMENT值。这个属性与C API中的`mysql_insert_id()`函数功能相当。  
这个属性也可以与语句句柄配合使用。
  - `$rv = $dbh->{ mysql_protoinfo };`  
返回与\$dbh相关联的数据库连接上所使用的客户/服务器协议版本，这个属性与C API中的`mysql_get_proto_info()`函数功能相当。
  - `$str = $dbh->{ mysql_serverinfo };`  
返回与\$dbh相关联的数据库连接上的服务器版本号，比如"4.0.2-alpha-log"。这个版本号由版本序号和一个或者多个后缀组成。这个属性所返回的信息与C API中的`mysql_get_server_info()`函数相同。各种版本号后缀的含义可以在附录C对`VERSION()`函数的介绍内容里查到。
  - `$str = $dbh->{ mysql_stat };`  
返回一个字符串，里面包含着服务器的运行状态信息。这个属性与C API中的`mysql_stat()`函数功能相当。
  - `$rv = $dbh->{ mysql_thread_id };`  
返回与\$dbh相关联的数据库连接的线程编号。这个属性与C API中的`mysql_thread_id()`函数功能相当。
- 从DBD::mysql 2.0900开始，有些MySQL特有的数据库句柄属性逐渐被一些新的属性所取代，所以大家在今后编写的脚本里应该尽量避免使用这些属性。这里把这些属性新、旧对照着列在了表G-2里。如果你现在使用的DBD::mysql驱动模块不支持新属性，不妨用那些旧属性试试，但最好的办法是把它升级到新版本。要是想让自己的脚本在新、旧版本的DBD::mysql模块下都能使用，就应该在脚本里使用一些如下所示的代码：

```
my $info = $dbh->{mysql_info};
# use old form if new form unavailable
$info = $dbh->{info} unless defined ($info);
```

表G-2 已逐渐退出使用的MySQL特有的数据库句柄属性

已逐渐退出使用的属性	推荐使用的属性
errno	mysql_errno
error	mysql_error
hostinfo	mysql_hostinfo
info	mysql_info
protoinfo	mysql_protoinfo
serverinfo	mysql_serverinfo
stats	mysql_stat
thread_id	mysql_thread_id

## G.4.4 语从句柄属性

语从句柄属性大都只适用于SELECT查询（或者其他与SELECT类似的查询），而且，在你调用prepare()方法获得一个语从句柄并通过这个句柄发出execute()调用之前，这些属性往往不可用。此外，因为finish()调用会使大多数语从句柄失效，所以在发出finish()调用之后（或者在已经到达结果集里的最后一个数据行之后——因为那些数据行取回函数在到达结果集末尾时会隐含地发出一个finish()调用）再去访问语从句柄属性是不明智的。

很多语从句柄属性的值是一个数组引用指针，这个数组里的元素分别对应着结果集里的一个数据行，而这个数组里的元素个数将由\$sth->{ NUM\_OF\_FIELDS }属性值给出。假设语句属性stmt\_attr就是一个这样的数组引用指针，那么，如果你需要把这个数组当做一个整体来访问，可以使用@{\$sth->{stmt\_attr}}形式的语法；如果你需要对这个数组里的各个元素依次进行处理，可以使用一个如下所示的循环语句：

```
for (my $i = 0; $i < $sth->{NUM_OF_FIELDS}; $i++)
{
    my $value = $sth->{stmt_attr}->[$i];
}
```

此外，对于NAME\_hash、NAME\_lc\_hash和NAME\_uc\_hash等几个属性返回散列引用指针，可以用一个如下所示的循环语句来遍历其中的各个元素：

```
foreach my $key (keys (%{$sth->{stmt_attr}}))
{
    my $value = $sth->{stmt_attr}->{$key};
}
```

- \$ary\_ref = \$sth->{ NAME };

这个属性返回的引用指针指向一个字符串数组，这个数组的各个元素依次给出了结果集里各个数据列的名称，它们的大小写情况与你在SELECT语句里所给出的一致。

- \$ary\_ref = \$sth->{ NAME\_hash };

这个属性返回的引用指针指向一个由字符串构成的散列值，各散列元素的键字就是结果集里各个数据列的名称，它们的大小写情况与你在SELECT语句里所给出的一致。散列元素的键值是与之对应的数据列在结果集数据行里的先后顺序（从0开始编号）。

NAME\_hash属性最早出现于DBI 1.20版本。

- \$ary\_ref = \$sth->{ NAME\_lc };

类似于NAME属性，但数据列的名称将全部返回为小写的字符串。

- \$ary\_ref = \$sth->{ NAME\_lc\_hash };

类似于NAME\_hash属性，但数据列的名称将全部返回为小写的字符串。

NAME\_lc\_hash属性最早出现于DBI 1.20版本。

- \$ary\_ref = \$sth->{ NAME\_uc };

类似于NAME属性，但数据列的名称将全部返回为大写的字符串。

- \$ary\_ref = \$sth->{ NAME\_uc\_hash };

类似于NAME\_hash属性，但数据列的名称将全部返回为大写的字符串。

NAME\_uc\_hash属性最早出现于DBI 1.20版本。

- `$ary_ref = $sth->{ NULLABLE };`

这个属性将返回一个数组引用指针，该数组的各个元素表明了与之对应的数据列是否允许使用NULL值，它们的取值可以是0（不允许使用NULL值）、1（允许使用NULL值）或2（不详）。

- `$rv = $sth->{ NUM_OF_FIELDS };`

结果集里的数据列个数。对于非SELECT语句，这个属性的返回值将是0。

- `$rv = $sth->{ NUM_OF_PARAMS };`

`prepare()`调用所处理的查询命令里有多少个占位符。

- `$ary_ref = $sth->{ PRECISION };`

这个属性将返回一个数组引用指针，该数组的各个元素表明了结果集里的各个数据列的精度。DBI使用的是ODBC对“精度”这个词的解释，它指的是MySQL数据列的最大宽度。对于数值数据列，这个“精度”其实只是它的显示宽度。对于字符串数据列，这个“精度”就是你在CREATE TABLE语句里给它定义的最大长度。

- `$ary_ref = $sth->{ SCALE };`

这个属性将返回一个数组引用指针，该数组的各个元素表明了结果集里的各个数据列的范围。DBI使用的是ODBC对“范围”这个词的解释，它指的是MySQL浮点数据列中小数点后面的位数。对于其他类型的数据列，这个属性所返回的“范围”值将等于0。

- `$str = $sth->{ Statement };`

与语句句柄`$sth`相关联的数据库查询语句的文本，即`prepare()`调用所看到的、尚未进行占位符替换的那个查询命令字符串。

- `$ary_ref = $sth->{ TYPE };`

这个属性将返回一个数组引用指针，该数组的各个元素表明了结果集里的各个数据列的类型。在DBD::mysql 1.1919之前的版本里，这个属性所返回的信息与`mysql_type`属性一样。但在DBD::mysql 1.1919及以后的版本里，这个属性的返回值里包含的是其他应用软件也可以理解和使用的数据类型编号，而`mysql_type`属性的返回值里则仍包含着MySQL专用的数据类型编号。

#### G.4.5 MySQL特有的语句句柄属性

本小节里的属性都是DBI MySQL驱动模块DBI::mysql所特有的。它们当中的绝大多数都应该只被当做只读属性来使用并应该在`execute()`调用完成之后再访问。但`mysql_store_result`和`mysql_use_result`属性却是两个例外——DBI::mysql模块将通过这两个属性来控制你的脚本对结果集进行处理的风格。DBI语句句柄属性`mysql_store_result`和`mysql_use_result`分别对应着C API函数`mysql_store_result()`和`mysql_use_result()`的结果集处理行为。这两个函数的具体用法和它们之间的区别详见附录F中的讨论。在默认的情况下，DBI将使用`mysql_store_result()`，但如果你激活了`mysql_use_result`属性，DBI就将使用`mysql_use_result()`。激活`mysql_use_result`属性的操作应该安排在`prepare()`调用之后、`execute()`调用之前进行，如下所示：

```
$sth = $dbh->prepare (...);
$sth->{mysql_use_result} = 1;
$sth->execute();
```



- `$rv = $sth->{ mysql_insertid };`  
与语句句柄\$sth相关联的连接上最近一次生成的AUTO\_INCREMENT值。  
这个属性也可以与数据库句柄配合使用。
- `$ary_ref = $sth->{ mysql_is_auto_increment };`  
这个属性将返回一个数组引用指针，该数组的各个元素表明了结果集里的各个数据列是否是AUTO\_INCREMENT数据列。  
这个属性最早出现于DBI::mysql 2.1014版本。
- `$ary_ref = $sth->{ mysql_is_blob };`  
这个属性将返回一个数组引用指针，该数组的各个元素表明了结果集里的各个数据列是否是BLOB或TEXT数据列。
- `$ary_ref = $sth->{ mysql_is_key };`  
这个属性将返回一个数组引用指针，该数组的各个元素表明了结果集里的各个数据列是否是某个键字的组成部分。
- `$ary_ref = $sth->{ mysql_is_num };`  
这个属性将返回一个数组引用指针，该数组的各个元素表明了结果集里的各个数据列是否是数值类型。
- `$ary_ref = $sth->{ mysql_is_pri_key };`  
这个属性将返回一个数组引用指针，该数组的各个元素表明了结果集里的各个数据列是否是某个PRIMARY KEY的组成部分。
- `$ary_ref = $sth->{ mysql_length };`  
类似于PRECISION属性。
- `$ary_ref = $sth->{ mysql_max_length };`  
这个属性将返回一个数组引用指针，该数组的各个元素给出了结果集各数据列里的数据值的实际最大长度。
- `$sth->{ mysql_store_result };`  
如果这个属性被激活（即被设置为1），DBI就将使用C API函数mysql\_store\_result()而不是mysql\_use\_result()到MySQL服务器上去检索结果集。这两个函数的具体用法和它们之间的区别详见附录F中的讨论。  
激活mysql\_store\_result属性的操作应该安排在prepare()调用之后、execute()调用之前进行。
- `$ary_ref = $sth->{ mysql_table };`  
这个属性将返回一个数组引用指针，该数组的各个元素表明了结果集里的各个数据列是来自哪一个数据表的。如果结果集里的某个数据列是某表达式的计算结果，它在这个数组里的对应元素就将是空字符串。
- `$ary_ref = $sth->{ mysql_type };`  
这个属性将返回一个数组引用指针，该数组的各个元素给出了结果集里的各个数据列的MySQL专用数据类型编号。
- `$ary_ref = $sth->{ mysql_type_name };`  
这个属性将返回一个数组引用指针，该数组的各个元素给出了结果集里的各个数据列的

MySQL专用数据类型名称。

- `$sth->{mysql_use_result}`;

如果这个属性被激活（即被设置为1），DBI就将使用C API函数`mysql_use_result()`而不是`mysql_store_result()`到MySQL服务器上去检索结果集。这两个函数的具体用法和它们之间的区别详见附录F中的讨论。

需要提醒大家注意的是，把这个属性设置为激活状态将使`mysql_max_length`等几个其他的属性失效。它还会使`row()`方法失效——虽然统计结果集有多少个数据行的工作应该用一边取回数据行一边对它们进行统计的办法来完成最好。

激活`mysql_use_result`属性的操作应该安排在`prepare()`调用之后、`execute()`调用之前进行。DBD::mysql旧版本里的某些MySQL特有的语从句柄属性逐渐被一些新的属性所取代，所以大家在今后编写的脚本里应该尽量避免使用这些属性。这里把这些属性新、旧对照着列在了表G-3里。如果你现在使用的DBD::mysql驱动模块不支持新属性，不妨用那些旧属性试试，但最好的办法是把它升级到新版本。要是想让自己的脚本在新、旧版本的DBD::mysql模块下都能使用，就应该在脚本里使用一些如下所示的代码：

```
my $lengths = $sth->{PRECISION};
# use old form if new form unavailable
$lengths = $sth->{length} unless defined ($lengths);
```

注意，`insertid`只是一个语从句柄属性，而它的替代品`mysql_insertid`却既可以被当做数据库句柄属性来使用，也可以被当做语从句柄属性来使用。

表G-3 已逐渐退出使用的MySQL特有的语从句柄属性

已逐渐退出使用的属性	推荐使用的属性
<code>insertid</code>	<code>mysql_insertid</code>
<code>is_blob</code>	<code>mysql_is_blob</code>
<code>is_key</code>	<code>mysql_is_key</code>
<code>is_not_null</code>	<code>NULLABLE</code>
<code>is_num</code>	<code>mysql_is_num</code>
<code>is_pri_key</code>	<code>mysql_is_pri_key</code>
<code>length</code>	<code>PRECISION</code>
<code>max_length</code>	<code>mysql_max_length</code>
<code>table</code>	<code>mysql_table</code>

#### G.4.6 动态属性

下面这些属性只与你最近一次使用的句柄（在以下内容里，将使用`$h`来代表这个句柄）相关联。如果你想正确地获得这些属性所返回的信息，就应该尽可能早地在发出前一个设置这些属性的DBI调用之后、在发出后一个会重新设置这些属性的DBI调用之前去访问它们。

- `$rv = $DBI::err;`

与调用`$h->err()`方法的效果相同。

- `$str = $DBI::errstr;`

与调用`$h->errstr()`方法的效果相同。

- `$rows = $DBI::rows;`

与调用`$h->rows()`方法的效果相同。

## G.5 DBI环境变量

表G-4列出了几个常用的DBI环境变量。这些环境变量（除DBI\_TRACE外）都是`connect()`方法可能会用到的。DBI\_DRIVER还可以用在`data_source()`方法里，而DBI\_TRACE是用于`trace()`方法里的。

表G-4 DBI环境变量

变 量 名	含 义
DBI_DRIVER	DBD级的驱动模块名称（对应于MySQL的驱动模块是mysql）
DBI_DSN	数据源名称
DBI_PASS	口令
DBI_TRACE	跟踪级别和/或跟踪输出文件
DBI_USER	用户名

## 附录H PHP API指南

本附录将对用来开发MySQL应用程序的PHP应用程序设计接口进行介绍。这个API由一组能够完成连接MySQL服务器和访问MySQL数据库等工作的函数构成。我们可以把这些函数划分为以下几大类：

- 连接管理类函数，用来建立和断开与MySQL服务器的连接。
- 出错报告类函数，用来获得出错代码和出错信息。
- 查询构造与执行类函数，用来构造数据库查询命令并把它们发送给MySQL服务器去执行。
- 结果集处理类函数，用来对从MySQL服务器返回的查询结果数据进行处理。
- 信息收集类函数，用来查知关于客户程序、MySQL服务器、协议版本以及当前连接的描述性信息。
- 正逐渐被淘汰的函数，即那些现在已经被认为是过时了的函数。

这里并不想把这篇指南写成一本教科书，所以其中收录的都是一些演示PHP API使用方法的代码片段。在本书的第8章里，大家可以看到一些完整的脚本以及编写它们的指导意见。PHP使用手册厚达数百页，但本附录只收录了那些与MySQL应用程序设计工作有直接关系的PHP函数，它们只是PHP语言很小的一个组成部分。如果想获得完整的PHP使用手册，请访问PHP的官方网站<http://www.php.net/>。

### H.1 编写PHP脚本

PHP脚本本身是一些普通的文本文件，但它的内容却是HTML和PHP代码的混合体。PHP脚本以解释方式执行，它们生成的输出就是将被发送给客户的Web页面。脚本中的HTML代码将不加解释地按原样发送出去，其中的PHP代码则会被解释执行并被替换为这段代码所产生的输出。

PHP将在HTML模式里开始读取一个脚本文件。当读到PHP代码部分的起始标记时，它就会从HTML模式切换到PHP代码执行模式并以解释方式执行有关的代码；等读到PHP代码部分的结束标记时，它又会从PHP代码执行模式切换回HTML模式。利用这些特殊的标记，可以在一个脚本文件里在HTML模式和PHP代码执行模式之间任意切换。PHP能够识别的起始/结束标记共有四种样式，但它们当中有些需要在事先明确地加以激活才能使用。激活起始/结束标记的办法之一是在PHP的初始化文件php.ini（PHP 3的初始化文件是php3.ini）里进行设置。这个文件的存放位置随着系统的不同而有所差异，在UNIX系统上，它通常在/usr/lib或/usr/local/lib目录里；在Windows系统上，它通常在Windows的系统目录里。

PHP能够识别以下几种样式的起始/结束标记：

- 默认样式，使用<?php和?>标记，如下所示：

```
<?php print ("Hello, world." ); ?>
```

- 简约样式，使用<?和?>标记，如下所示：

```
<? print ("Hello, world." ); ?>
```

在PHP 4里，如果你只是想把某个表达式的结果打印出来，还可以省略print命令并把这对标记写成<?=和?>形式，如下所示：

```
<?= "Hello, world." ?>
```

简约样式可以通过PHP初始化文件里的下面这条指令激活：

```
short_open_tag = On ;
```

- ASP (Active Server Page) 兼容样式，使用<%和%>标记，如下所示：

```
<% print ("Hello, world." ); %>
```

在PHP 4里，如果你只是想把某个表达式的结果打印出来，还可以省略print命令并把这对标记写成<%=和%>形式，如下所示：

```
<%= "Hello, world." %>
```

ASP兼容样式可以通过PHP初始化文件里的下面这条指令激活：

```
asp_tags = On ;
```

对ASP兼容标记的支持最早出现于PHP 3.0.4版本。

- 如果你使用的HTML编辑器不支持上述风格的标记，可以使用<script>和</script>标记，如下所示：

```
<script language = "php"> print ("Hello, world." ); </script>
```

本附录中的脚本示例将使用<?php和?>作为PHP脚本代码的起始标记和结束标记。

## H.2 函数

从现在开始，我们将依次对PHP语言中与MySQL有关的各个函数进行介绍。但在展开讨论之前，这里先对后面内容里的常见参数做一下说明：

- **conn\_id**：连接标识符。本节中的大多数函数都有连接标识符参数。这是一个用来标识某个MySQL服务器连接的resource数据项，通常需要通过mysql\_connect()调用或mysql\_pconnect()调用获得。一般说来，如果某个函数的连接标识符参数是可选的而你在调用该函数时没有给出这个参数，PHP就会使用你最近一次打开的MySQL服务器连接。在你没有给出连接标识符参数或者该连接尚未打开的情况下，很多函数会先尝试建立这个连接，然后再去完成相应的操作动作。
- **result\_id**：结果集标识符。数据库查询命令的查询结果是一些数据行，这些数据行将构成一个集合，即结果集。用来标识这些结果集的resource数据项就叫做结果集标识符。有几个函数可以用来创建结果集，其中最常见的是mysql\_query()调用。
- **row\_num**和**col\_num**：这两个参数分别给出了结果集里的数据行和数据列的个数/编号。它们都是从0开始计算的。
- 在本附录里，我们将把各有关PHP函数的可选参数放在方括号（[]）里。



在执行出错的时候，除返回一个状态值以外，有些函数还会产生一条出错信息。在Web上下文里，这条信息将随脚本的其他输出一起被发送到客户的Web浏览器去，但这可能并不符合你的希望。解决这一问题的办法之一是在函数名称的前面加上一个@字符。请看下面这段示例代码：因为加上了@字符，所以这段代码在mysql\_connect()调用执行出错的时候不会把它产生的出错信息发送给客户，客户只能在Web浏览器里看到你事先安排的“Could not connect”信息，这种出错处理方式更符合我们的意愿：

```
<?php
    $conn_id = @mysql_connect ("cobra.snake.net", "sampadm", "secret")
        or die ("Could not connect\n");
    print ("Connected successfully\n");
?>
```

本附录里的示例代码将全部使用@字符来抑制连接PHP函数本身产生的出错信息，如果真的执行出错，脚本代码将显示由我们安排的出错报告消息。

抑制出错信息的另一种办法是使用error\_reporting()函数，如下所示：

```
<?php
    error_reporting (0); # suppress all error messages
    $conn_id = mysql_connect ("cobra.snake.net", "sampadm", "secret")
        or die ("Could not connect\n");
    print ("Connected successfully\n");
?>
```

在成功地连接上MySQL服务器之后，本附录中的示例脚本大都会显示一条“Connected successfully”消息。做出这种安排的目的是为了让这些脚本（在你自己去试用它们的时候）能够在连接成功时输出一些东西。（在某些浏览器里，一个执行成功但没有产生任何输出的PHP脚本往往会导致显示一条“page contains no data”（本页面没有数据）警告消息，而这可能会让你误认为脚本在什么地方出了错。）

本附录里的示例脚本在显示消息或查询结果的时候通常都把它们输出为普通文本。做出这种安排的目的是为了增加这些示例脚本的可阅读性。但在那些将在Web环境里执行的脚本里，应该先用htmlspecialchars()函数对输出信息做一下处理，这样，那些在HTML里有特殊含义或用途的字符（如<、>或&等）就可以正确地显示在客户的浏览器里了。

在下面的内容里，“SELECT查询”代表着能够返回一些数据行的各种数据库查询命令，比如SELECT、DESCRIBE、EXPLAIN或SHOW等等。

### H.2.1 连接管理类函数

本小节向大家介绍用来打开/关闭与MySQL服务器的连接以及改变MySQL用户名的函数。

• int

```
mysql_change_user (string user_name,
                    string password
                    [, string db_name
                    [, resource conn_id ] ] );
```

改变MySQL服务器连接conn\_id上的用户名和默认数据库, 如果没有给出conn\_id参数, 则使用当前连接。在发出这个调用之后, 如果你在给出数据表名字的时候没有在它们的前面加上一个数据库名进行指定, 就表示你想使用的是默认数据库里的数据表。

如果user\_name参数指定的用户有权连接到该服务器且有权访问db\_name参数指定的数据库的话(如果给出db\_name参数), mysql\_change\_user()将返回true(真); 否则, 这次调用将失败, 当前用户和当前数据库保持不变。

mysql\_change\_user()要求MySQL的版本不低于3.23.3。它最早出现于PHP 3.0.13版本, 但在PHP 4里却又被取消了。

```
<?php
    $conn_id = @mysql_connect ("cobra.snake.net", "sampadm", "secret")
        or die ("Could not connect\n");
    mysql_change_user ("unknown", "public")
        or die ("Could not change user\n");
    print ("User changed successfully\n");
?>
```

#### • bool

**mysql\_close** ([ resource conn\_id ] );

关闭由conn\_id参数指定的MySQL服务器连接。如果没有给出conn\_id参数, mysql\_close()调用将关闭最近一次打开的服务器连接。

如果调用成功, mysql\_close()将返回true; 如果执行出错, 则将返回false。如果conn\_id参数指定的是一条由mysql\_pconnect()调用打开的永久性连接, mysql\_close()将返回true但该连接仍将保持打开状态。如果你事先已经准备关闭某条与MySQL服务器的连接, 就应该使用mysql\_connect()函数而不是mysql\_pconnect()函数去打开它。

```
<?php
    $conn_id = @mysql_connect ("cobra.snake.net", "sampadm", "secret")
        or die ("Could not connect\n");
    print ("Connected successfully\n");
    mysql_close ($conn_id);
?>
```

非永久性连接并不需要由你明确地通过一个mysql\_close()调用去关闭, PHP会在脚本执行结束的时候自动关闭它们。但如果你的脚本在完成MySQL服务器的访问操作后还需要执行一段时间, 就应该及时地把不再需要使用的服务器连接关闭掉。这是一种良好的编程习惯, 对MySQL服务器也很有好处, 因为这将使它能够以有限的资源尽可能多地向其他人提供服务。

#### • resource

**mysql\_connect** ( [ string host\_name ]  
 [, string user\_name  
 [, string password  
 [, new\_conn ] ] ] );

以user\_name参数指定的用户名、password参数指定的口令去连接由参数host\_name指定的主机上的MySQL服务器。如果调用成功，这个函数将返回一个与新建连接相关联的连接标识符；如果执行出错，则返回false。

从PHP 3.0B4开始，主机名参数还允许以"host\_name:port\_num"的格式给出，即允许再指定一个可选的连接端口号。从PHP 3.0.10开始，如果主机名是"localhost"，那么主机名参数还允许以"localhost:socket\_name"的格式给出，即允许再指定一个可选的UNIX域套接字文件路径名，套接字文件应该以完整的路径名形式给出。如果想使用TCP/IP连接而不是通过UNIX套接字来建立连接，请把主机名参数值设定为"127.0.0.1"。

如果没有给出主机名参数，则该参数的默认值将是"localhost"。如果用户名参数缺失或者为空，该参数的默认值将是用来运行PHP的用户名：如果PHP被运行作为一个Apache模块，这个默认用户名就是用来运行Web服务器（即Apache）的那个账户的用户名；如果PHP被运行作为一个独立的程序，这个默认用户名就将是用来运行PHP脚本的那个用户名。如果口令参数缺失或者为空，则将以空字符串作为默认的口令。

在PHP 4里，可以在PHP的初始化文件php.ini里预先为这个函数中缺失的各有关参数设定一个默认值。表H-1列出了相应的配置指令。

表H-1 PHP 4用来设定默认连接参数的配置指令

连接参数	配置指令
主机名	mysql.default_host
用户名	mysql.default_user
口令	mysql.default_password
端口号	mysql.default_port
套接字路径名	mysql.default_socket

在连接打开期间，如果脚本以完全相同的连接参数（主机名、用户名、口令）再次调用mysql\_connect()函数，PHP将不会建立一个新的与MySQL服务器的连接——第二个mysql\_connect()调用将简单地返回那个现有的连接标识符。但从PHP 4.2.0开始，如果在第二次调用mysql\_connect()函数的时候把它的new\_conn参数设置为true，就会强制它去建立一个新的连接。

mysql\_connect()所建立的连接可以通过mysql\_close()调用来关闭。如果某个连接等到脚本结束执行时仍未被关闭，PHP将自动关闭它们。

很多PHP函数中的连接标识符参数conn\_id都是可选的。如果没有给出这个参数，那些函数通常就会使用最近一次打开的连接。如果当时没有已经打开的连接，有些函数（比如那些用来发出数据库查询命令的函数、用来选择/创建/丢弃数据库的函数、用来检索关于连接本身的信息的函数等等）将先去尝试建立一个连接，然后再开始执行它们的“本职工作”。在这几种情况里，脚本的执行效果都相当于先对mysql\_connect()函数做了一次无参数调用，即先使用各有关参数的默认值进行了一次mysql\_connect()调用。如果这种隐含的连接尝试失败了，引发这种隐含调用的那个函数也将失败。

```
<?php
    $conn_id = @mysql_connect ("cobra.snake.net", "sampadm", "secret")
        or die ("Could not connect\n");
    print ("Connected successfully\n");
    mysql_close ($conn_id);
?>
```

- resource

**mysql\_pconnect** ( [ string host\_name ]  
 [, string user\_name  
 [, string password ] ] );

mysql\_pconnect()与mysql\_connect()只有一个区别：前者试图建立的是永久性连接，这个连接将保持打开状态直到脚本结束执行。在此期间，如果脚本以完全相同的连接参数（主机名、用户名、口令）再次调用mysql\_pconnect()函数，PHP将不会去建立一个新的与MySQL服务器的连接——第二个mysql\_pconnect()调用将简单地返回那个现有的连接标识符。因为没有断开和再次建立有关连接的开销，所以永久性连接通常要比非永久性连接的效率更高。

不过，永久性连接只有在PHP被运行于Web服务器进程内的一个需要在PHP脚本结束执行后继续运行的模块时才有实际意义——当Web服务器再次接收某个PHP脚本以完全相同的连接参数去连接MySQL的请求时，就可以简单地重复使用这个永久性连接而不需要重新建立一个连接。反之，如果脚本是由一个独立的PHP进程去执行的，那么，当这个脚本结束执行的时候，由它建立的永久性连接也都将关闭，因为那个PHP进程将随着脚本的执行结束而结束运行。

试图通过发出一个mysql\_close()调用来关闭一条永久性连接的举动是毫无意义的——虽然mysql\_close()会返回true，但那条永久性连接却仍将保持打开状态。

```
<?php
    $conn_id = @mysql_pconnect ("cobra.snake.net", "sampadm", "secret")
        or die ("Could not connect\n");
    print ("Connected successfully\n");
?>
```

- bool

**mysql\_ping** ( [ resource conn\_id ] );

在连接上MySQL服务器之后，如果脚本长时间没有进行数据库操作，服务器就会在一段倒计时时间结束后自动断开这个连接。mysql\_ping()调用将重新建立这条连接。如果连接仍处于连通状态或者成功地重新连接上了MySQL服务器，mysql\_ping()将返回true；否则，将返回false。mysql\_ping()函数最早出现于PHP 4.3.0版本。

## H.2.2 出错报告类函数

mysql\_errno()和mysql\_error()函数负责返回与MySQL有关的PHP函数的出错代码和出错报

告信息。不过，在PHP 4.0.6之前，如果连接标识符本身无效的话，这两个函数就都无法返回有用的出错信息。这意味着我们无法通过这两个函数去了解mysql\_connect()或mysql\_pconnect()调用的失败原因——因为没有成功的mysql\_connect()或mysql\_pconnect()调用无法返回一个有效的连接标识符。在这种场合，如果你想知道你的脚本为什么没有成功地连接上MySQL服务器，就需要另想办法：先在PHP初始化文件php.ini里用下面这条指令激活track\_errors变量：

```
track_errors =On ;
```

再重新启动你的Web服务器（如果你把PHP运行作为一个Apache模块的话）。完成这些准备工作之后，就可以在脚本里通过\$php\_errormsg变量给出的字符串去了解为什么你的脚本没能连接上MySQL服务器了，如下所示：

```
<?php
    $conn_id = @mysql_connect("badhost", "baduser", "badpass")
        or die ("Could not connect: $php_errormsg\n");
    print ("Connected successfully\n");
?>
```

我们可以在PHP脚本里把这两种机制结合起来使用，即如果出错报告类例程能够返回关于连接失败的原因的信息，则使用出错报告例程；如果它们不能返回这些信息，则使用\$php\_errormsg变量。下面这段示例代码演示了这一思路的具体实现办法：如果mysql\_error()函数的返回值非零，则表示可以通过出错报告例程来了解连接失败的原因；否则，就通过\$php\_errormsg变量来查知出错的原因。如下所示：

```
<?php
    if (!($conn_id = @mysql_connect("badhost", "baduser", "badpass")))
    {
        # mysql_errno() returns non-zero if it works for connect errors
        if (mysql_errno ())
        {
            $msg = sprintf ("Could not connect: %s (%d)\n",
                            mysql_error (), mysql_errno ());
        }
        else # fall back to $php_errormsg
        {
            $msg = "Could not connect: $php_errormsg\n";
        }
        die ($msg);
    }
    print ("Connected successfully\n");
?>
```

• int

**mysql\_errno** ([ resource conn\_id ] );

给定一个连接（若没有给出conn\_id参数，则使用当前连接），这个函数将返回脚本在该连接上最近一次调用的与MySQL有关的PHP函数的出错代码（注意，有些PHP函数不返回状态信息）。若返回值是0，则表示没有出错。



```

<?php
    $conn_id = @mysql_connect ("cobra.snake.net", "sampadm", "secret")
        or die ("Could not connect, error=" . mysql_errno () . "\n");
    print ("Connected successfully\n");
    mysql_select_db ("sampdb")
        or die ("Could not select database, error=" . mysql_errno () . "\n");
    $query = "SELECT * FROM president";
    $result_id = mysql_query ($query)
        or die ("query failed, error=" . mysql_errno () . "\n");
    mysql_free_result ($result_id);
?>

```

- string

**mysql\_error** ([ resource conn\_id ] );

给定一个连接（若没有给出conn\_id参数，则使用当前连接），这个函数将返回脚本在该连接上最近一次调用的与MySQL有关的PHP函数的出错信息字符串（注意，有些PHP函数不返回状态信息）。若返回值是一个空字符串，则表示没有出错。

```

<?php
    $conn_id = @mysql_connect ("cobra.snake.net", "sampadm", "secret")
        or die ("Could not connect, error=" . mysql_error () . "\n");
    print ("Connected successfully\n");
    mysql_select_db ("sampdb")
        or die ("Could not select database, error=" . mysql_error () . "\n");
    $query = "SELECT * FROM president";
    $result_id = mysql_query ($query)
        or die ("query failed, error=" . mysql_error () . "\n");
    mysql_free_result ($result_id);
?>

```

### H.2.3 查询构造与执行类函数

本小节里的函数负责构造并向MySQL服务器发出数据库查询命令。在PHP脚本里，每个查询命令字符串只能包含一条SQL语句，并且不允许在末尾加上分号(;)或\g结束符；结束符“;”和\g都是mysql客户程序使用的表示法，PHP脚本所发出的数据库查询命令不需要它们来表示SQL语句的结束。

- string

**mysql\_escape\_string** ( string str );

对str参数指定的字符串里的引号或其他特殊字符进行转义。它将在特殊字符的前面加上一个反斜线字符(\)并把经如此处理而得到的新字符串作为返回值。把经过这个函数处理的数据值插入到SQL查询命令能够保证它们不会引起语法错误。PHP 4.3.0及以后的版本还支持一个名为mysql\_real\_escape\_string()的函数，那个函数的用途和用法与这里的mysql\_escape\_string()函数是一样的，但那个函数增加了一个连接标识符输入参数并在对字符串str进行转义编码时把当前字符集也作为一项考虑因素，详细情况请参见本小节后面内

容里的mysql\_real\_escape\_string()条目。

可以在本书的附录F对C API函数mysql\_real\_escape\_string()的介绍内容里找到一份需要进行转义处理的SQL特殊字符的清单。

mysql\_escape\_string()函数最早出现于PHP 4.0.3版本。在此之前，可以通过addslashes()函数获得相同的效果。

```
<?php
    $conn_id = @mysql_connect ("cobra.snake.net", "sampadm", "secret")
        or die ("Could not connect\n");
    mysql_select_db ("sampdb")
        or die ("Could not select database\n");
    $last_name = mysql_escape_string ("O'Malley");
    $first_name = mysql_escape_string ("Brian");
    $query = "INSERT INTO member (last_name,first_name,expiration)"
        . " VALUES('$last_name','$first_name','2002-6-3')";
    $result_id = mysql_query ($query)
        or die ("Query failed\n");
    printf ("membership number for new member: %d\n",
        mysql_insert_id());
?>
```

#### • resource

**mysql\_list\_dbs** ([ resource conn\_id ] );

如果调用成功，这个函数将返回一个结果集标识符，该结果集由给定MySQL服务器所知道的数据库名称构成，结果集里的每一行对应着一个数据库。如果执行出错，则返回false。在调用这个函数之前，脚本不必选定默认数据库。这个结果集可以用任何一个数据行取回函数或者mysql\_db\_names()函数来处理。需要提醒大家注意的是，如果用来处理这个结果集的某个函数需要一个数据行序号来作为其输入参数，该序号必须落在从0到mysql\_num\_rows()-1的范围内。还可以把这个函数所返回的结果集标识符传递给mysql\_free\_result()函数以释放与之关联的各种资源。

```
<?php
    $conn_id = @mysql_connect ("cobra.snake.net", "sampadm", "secret")
        or die ("Could not connect\n");
    $result_id = mysql_list_dbs ()
        or die ("Query failed\n");
    print ("Databases (using mysql_fetch_row()):<br />\n");
    while ($row = mysql_fetch_row ($result_id))
        printf ("%s<br />\n", $row[0]);
    mysql_free_result ($result_id);
    $result_id = mysql_list_dbs ()
        or die ("Query failed\n");
    print ("Databases (using mysql_db_name()):<br />\n");
    for ($i = 0; $i < mysql_num_rows ($result_id); $i++)
        printf ("%s<br />\n", mysql_db_name ($result_id, $i));
    mysql_free_result ($result_id);
?>
```

## • resource

```
mysql_list_fields ( string db_name,  
                    string tbl_name  
                    [, resource conn_id ] );
```

如果调用成功，这个函数将返回一个结果集标识符，该结果集包含着对给定数据表（这个数据表由db\_name和tbl\_name参数共同指定）里的各数据列的描述信息，如果执行出错，则返回false。在调用这个函数之前，脚本不必选定默认数据库。把这个函数所返回的结果集标识符用做mysql\_field\_flags()、mysql\_field\_len()、mysql\_field\_name()和mysql\_field\_type()等函数的输入参数，就能进一步获取相应的具体信息。还可以把这个函数所返回的结果集标识符传递给mysql\_free\_result()函数以释放与之关联的各种资源。

```
<?php  
$conn_id = @mysql_connect ("cobra.snake.net", "sampadm", "secret")  
    or die ("Could not connect\n");  
$result_id = mysql_list_fields ("sampdb", "member")  
    or die ("Query failed\n");  
print ("member table column information:<br />\n");  
for ($i = 0; $i < mysql_num_fields ($result_id); $i++)  
{  
    printf ("column %d:", $i);  
    printf (" name %s,", mysql_field_name ($result_id, $i));  
    printf (" len %d,", mysql_field_len ($result_id, $i));  
    printf (" type %s,", mysql_field_type ($result_id, $i));  
    printf (" flags %s\n", mysql_field_flags ($result_id, $i));  
    print ("<br />\n");  
}  
mysql_free_result ($result_id);  
?>
```

## • resource

```
mysql_list_processes ( [ resource conn_id ] );
```

如果调用成功，这个函数将返回一个结果集标识符，该结果集包含着正在指定MySQL服务器里运行着的各有关进程的描述信息，如果执行出错，则返回false。如果你拥有PROCESS 权限，输出清单将包含所有的服务器进程；如果你没有PROCESS 权限，输出清单将只包含那些属于你本人的进程。

这个函数所返回的结果集可以用任何一个数据行取回函数来处理。还可以把这个函数所返回的结果集标识符传递给mysql\_free\_result()函数以释放与之关联的各种资源。

mysql\_list\_processes()函数最早出现于PHP 4.3.0版本。

## • resource

```
mysql_list_tables ( string db_name  
                    [, resource conn_id ] );
```

如果调用成功，这个函数将返回一个结果集标识符，该结果集由参数db\_name所指定的数据库里的数据表名称构成，结果集里的每一行对应着一个数据表。如果执行出错，则返回

false。在调用这个函数之前，脚本不必选定默认数据库。这个结果集可以用任何一个数据行取回函数或者mysql\_tablename()函数来处理。需要提醒大家注意的是，如果用来处理这个结果集的某个函数需要一个数据行序号来作为其输入参数，该序号必须落在从0到mysql\_num\_rows()-1的范围内。还可以把这个函数所返回的结果集标识符传递给mysql\_free\_result()函数以释放与之关联的各种资源。

```
<?php
    $conn_id = @mysql_connect ("cobra.snake.net", "sampadm", "secret")
        or die ("Could not connect\n");
    $result_id = mysql_list_tables ("sampdb")
        or die ("Query failed\n");
    print ("sampdb tables (using mysql_fetch_row()):<br />\n");
    while ($row = mysql_fetch_row ($result_id))
        printf ("%s<br />\n", $row[0]);
    mysql_free_result ($result_id);
    $result_id = mysql_list_tables ("sampdb")
        or die ("Query failed\n");
    print ("sampdb tables (using mysql_tablename()):<br />\n");
    for ($i = 0; $i < mysql_num_rows ($result_id); $i++)
        printf ("%s<br />\n", mysql_tablename ($result_id, $i));
    mysql_free_result ($result_id);
?>
```

#### • int

**mysql\_query** (string query  
[, resource conn\_id  
[, int fetch\_mode ]]);

把查询命令字符串发送给MySQL服务器去执行。对于DELETE、INSERT、REPLACE或者UPDATE等SQL语句，如果调用成功，mysql\_query()将返回true；如果执行出错，则将返回false。在查询成功的前提下，还可以调用mysql\_affected\_rows()函数来查知该查询实际影响（即删除、插入、替换或者更新）了多少个数据行。

从PHP 4.2.0版本开始，这个函数新增了一个可选参数fetch\_mode，这个参数将决定mysql\_query()调用是立刻取回并缓存结果集还是等稍后再去取回结果集。fetch\_mode参数有MYSQL\_STORE\_RESULT（缓存）和MYSQL\_USE\_RESULT（不缓存）两个可取值，它在这个函数里的默认值是MYSQL\_STORE\_RESULT。如果把这个参数设置为MYSQL\_USE\_RESULT，mysql\_query()调用的操作行为将与mysql\_unbuffered\_query()调用相同，详细情况见mysql\_unbuffered\_query()条目。

对于SELECT语句，如果调用成功，mysql\_query()函数将返回一个正的结果集标识符；如果执行出错，则将返回false。在查询成功的前提下，可以把mysql\_query()调用所返回的结果集标识符传递给任何一个允许以result\_id为输入参数的结果集处理函数去做进一步的处理。还可以把这个函数所返回的结果集标识符传递给mysql\_free\_result()函数以释放与之关联的各种资源。

注意，所谓“查询成功”指的是查询命令在没有出错的情况下得以执行完毕，但这并不意味着查询命令必然会检索出一些数据行来。以下面这条查询命令为例，虽然它完全合法，却连一个数据行也检索不出来：

```
SELECT * FROM president WHERE 1 = 0
```

导致查询失败的原因有很多种，其中比较常见的有：1) 查询命令本身有语法错误；2) 查询命令虽然语法正确，但语义不正确（比如你试图选取的某个数据列在指定的数据表里根本就不存在等）；3) 你没有执行这条查询命令所需要的权限；4) 无法连接MySQL服务器（比如网络出现故障等）。

```
<?php
    $conn_id = @mysql_connect ("cobra.snake.net", "sampadm", "secret")
        or die ("Could not connect\n");
    print ("Connected successfully\n");
    mysql_select_db ("sampdb")
        or die ("Could not select database\n");
    $query = "SELECT * FROM president";
    $result_id = mysql_query ($query)
        or die ("Query failed\n");
    mysql_free_result ($result_id);
?>
```

#### • string

**mysql\_real\_escape\_string** ( string str  
[, resource conn\_id ] );

对str参数指定的字符串里的引号或其他特殊字符进行转义。它将在特殊字符的前面加上一个反斜线字符(\)并把经如此处理而得到的新字符串作为返回值。把经过这个函数处理的数据值插入到SQL查询命令能够保证它们不会引起语法错误。这个函数与mysql\_escape\_string()函数的用途和用法是一样的，但增加了一个连接标识符输入参数并在对字符串str进行转义编码时把当前字符集也作为一项考虑因素，mysql\_escape\_string()函数则做不到这一点。请参见本小节前面内容里的mysql\_escape\_string()条目。可以在本书的附录F对C API函数mysql\_real\_escape\_string()的介绍内容里找到一份需要进行转义处理的SQL特殊字符的清单。

mysql\_real\_escape\_string()要求MySQL的版本不低于3.23.14。它最早出现于PHP 4.3.0版本。在此之前，可以通过mysql\_escape\_string()或addslashes()函数获得同样的效果。

```
<?php
    $conn_id = @mysql_connect ("cobra.snake.net", "sampadm", "secret")
        or die ("Could not connect\n");
    mysql_select_db ("sampdb")
        or die ("Could not select database\n");
    $last_name = mysql_real_escape_string ("O'Malley");
    $first_name = mysql_real_escape_string ("Brian");
    $query = "INSERT INTO member (last_name,first_name,expiration)"
```



```

        . " VALUES('$last_name','$first_name','2002-6-3')";
$result_id = mysql_query ($query)
    or die ("Query failed\n");
printf ("membership number for new member: %d\n", mysql_insert_id());
?>

```

- bool

**mysql\_select\_db** ( string db\_name  
[, resource conn\_id ] );

把指定数据库选定为指定连接上的默认数据库。此后，如果在给出数据表名字的时候没有在它们的前面加上一个数据库名进行指定，就表示你想使用的是这个默认数据库里的数据表。如果调用成功，这个函数将返回true；如果执行出错，则将返回false。

```

<?php
$conn_id = @mysql_connect ("cobra.snake.net", "sampadm", "secret")
    or die ("Could not connect\n");
mysql_select_db ("sampdb")
    or die ("Could not select database\n");
print ("Database selected successfully\n");
?>

```

- int

**mysql\_unbuffered\_query** ( string query  
[, resource conn\_id  
[, int fetch\_mode ] ] );

这个函数与mysql\_query()很相似，但在默认的情况下，mysql\_unbuffered\_query()不会把结果集里的数据行立刻取回到内存里来。这两个函数在行为上的差异与C API函数mysql\_store\_result()和mysql\_use\_result()之间的差异（详见附录F中的讨论）是相对应的，附录F中的相关讨论也同样适用于PHP函数mysql\_query()和mysql\_unbuffered\_query()。需要特别说明的是，有些PHP函数（比如mysql\_data\_seek()、mysql\_num\_rows()、mysql\_result()以及它的别名函数等等）只有在数据库查询命令是在缓存模式里被发出的场合才能正确工作。从PHP 4.2.0版本开始，这个函数新增了一个可选参数fetch\_mode，这个参数将决定mysql\_unbuffered\_query()调用是立刻取回并缓存结果集还是等稍后再去取回结果集。fetch\_mode参数有MYSQL\_STORE\_RESULT（缓存）和MYSQL\_USE\_RESULT（不缓存）两个可取值，它在这个函数里的默认值是MYSQL\_USE\_RESULT。在非缓存取回模式里，必须把前一条查询命令所返回的数据行全部取回之后才能成功地发出下一条数据库查询命令。可以把这个函数所返回的结果集标识符传递给mysql\_free\_result()函数以释放与之关联的各种资源——它将自动地替你检索并丢弃所有尚未被取回的数据行。

mysql\_unbuffered\_query()函数最早出现于PHP 4.0.6版本。

```

<?php
$conn_id = @mysql_connect ("cobra.snake.net", "sampadm", "secret")
    or die ("Could not connect\n");

```

```

mysql_select_db ("sampdb")
    or die ("Could not select database\n");
$query = "SELECT first_name, last_name FROM member";
$result_id = mysql_unbuffered_query ($query)
    or die ("Query failed\n");
while (list ($first_name, $last_name)
        = mysql_fetch_row ($result_id))
    print ("$first_name $last_name\n");
mysql_free_result ($result_id);
?>

```

## H.2.4 结果集处理类函数

本小节里的函数负责对数据库查询命令的结果集进行检索。它们还能让你访问到与结果集有关的其他一些信息，比如有多少数据行受到了影响、结果集里的各数据列的元数据等等。

- int

**mysql\_affected\_rows** ([ resource conn\_id ] );

如果调用成功，这个函数将返回指定连接上最近一次DELETE、INSERT、REPLACE或UPDATE查询所实际影响（即删除、插入、替换或者更新）的数据行个数，0表示没有数据行受到影响；如果执行出错，则将返回-1。

如果mysql\_affected\_rows()调用是在一次SELECT查询之后发出的，它将返回该查询实际检索到的数据行个数。但一般说来，还是应该用mysql\_num\_rows()函数去查知SELECT查询实际检索到的数据行个数。

```

<?php
$conn_id = @mysql_connect ("cobra.snake.net", "sampadm", "secret")
    or die ("Could not connect\n");
mysql_select_db ("sampdb")
    or die ("Could not select database\n");
$query = "INSERT INTO member (last_name,first_name,expiration)"
    . " VALUES('Brown','Marcia','2002-6-3')";
$result_id = mysql_query ($query)
    or die ("Query failed\n");
$count = mysql_affected_rows ();
printf ("%d row%s inserted\n", $count, $count == 1 ? "" : "s");
?>

```

- bool

**mysql\_data\_seek** ( resource result\_id, int row\_num );

每一个SELECT查询所返回的结果集里都有一个供内部使用的数据行指针，这个指针的作用是告诉数据行取回函数mysql\_fetch\_array()、mysql\_fetch\_assoc()、mysql\_fetch\_object()或mysql\_fetch\_row()等在下次被调用时应该返回哪一个数据行。如果想把结果集里的这个内部指针设置为指向某个特定的数据行，就需要调用mysql\_data\_seek()函数来达到这一

目的。数据行序号参数row\_num必须落在从0到mysql\_num\_rows()-1的范围内。如果数据行序号参数合法且调用成功，这个函数将返回true；否则，将返回false。

```
<?php
    $conn_id = @mysql_connect ("cobra.snake.net", "sampadm", "secret")
        or die ("Could not connect\n");
    mysql_select_db ("sampdb")
        or die ("Could not select database\n");
    $query = "SELECT last_name, first_name FROM president";
    $result_id = mysql_query ($query)
        or die ("Query failed\n");
    # fetch rows in reverse order
    for ($i = mysql_num_rows ($result_id) - 1; $i >= 0; $i--)
    {
        if (!mysql_data_seek ($result_id, $i))
        {
            printf ("Cannot seek to row %d\n", $i);
            continue;
        }
        if (!$row = mysql_fetch_object ($result_id))
            continue;
        printf ("%s %s<br />\n", $row->last_name, $row->first_name);
    }
    mysql_free_result ($result_id);
?>
```

#### • string

**mysql\_db\_name** ( resource result\_id,  
int row\_num  
[, mixed field ] );

给定一个由mysql\_list\_dbs()函数所返回的结果集标识符和该结果集里的一个数据行序号，如果调用成功，mysql\_db\_name()函数将返回保存在该结果集指定数据行的数据库的名称；如果执行出错，则将返回false。数据行序号参数row\_num必须落在从0到mysql\_num\_rows()-1的范围内。

mysql\_db\_name()函数最早出现于PHP 3.0.6版本。它实际上是mysql\_result()函数的别名。

```
<?php
    $conn_id = @mysql_connect ("cobra.snake.net", "sampadm", "secret")
        or die ("Could not connect\n");
    $result_id = mysql_list_dbs ()
        or die ("Query failed\n");
    print ("Databases:<br />\n");
    for ($i = 0; $i < mysql_num_rows ($result_id); $i++)
        printf ("%s<br />\n", mysql_db_name ($result_id, $i));
    mysql_free_result ($result_id);
?>
```

## • array

**mysql\_fetch\_array** ( resource result\_id  
[, int result\_type ] );

如果调用成功，这个函数将把给定结果集里的下一个数据行返回为一个数组；如果已经到达结果集里的最后一个数据行，则将返回false。这个数组里的元素既可以通过各有关数据列在结果集里的序号（从0开始）访问，也可以通过各有关数据列的名称访问。换句话说，这个数组有两套下标，一套是由数据列序号充当的数值下标，另一套是由数据列名称来充当的关联键字（即字符串下标），而两套下标都能让你访问到各有关数据列里的数据值。关联键字需要区分字母的大小写情况，所以必须按与你在有关的数据库查询命令里所使用的数据列名称完全一致的形式来给出它们。比如说，假设发出了一条如下所示的数据库查询命令：

```
SELECT last_name, first_name FROM president
```

那么，在把结果集里的数据行取回到一个名为\$row的数组里之后，就可以像下面这样来访问该数组里的各个元素了：

```
$row[0]           Holds last_name value
$row[1]           Holds first_name value
$row["last_name"] Holds last_name value
$row["first_name"] Holds first_name value
```

需要提醒大家注意的是，键字本身只包含数据列的名字而没有数据表的名字。因此，如果某个数据库查询命令从不同的数据表里选取了一些同名的数据列，它们就会在结果集里相互“遮盖”，而以该数据列的名称作为键字将只能检索到在该查询命令所选取的数据列清单里最后一个出现的同名数据列。要想访问那些被“遮盖”的数据列，就必须使用它们的数值下标；或者重新改写这条查询命令，给那些同名的数据列分别取一个独一无二的别名。在PHP 3.0.7之前，mysql\_fetch\_array()函数总是同时返回一套数值下标和一套关联键字。但从PHP 3.0.7开始，可以利用这个函数新增的result\_type参数来控制它返回哪一套下标。这个result\_type参数有MYSQL\_ASSOC（只返回关联键字）、MYSQL\_NUM（只返回数值下标）和MYSQL\_BOTH（同时返回两套下标）三种可取值。如果result\_type参数缺失，其默认值将是MYSQL\_BOTH；如果result\_type参数是MYSQL\_ASSOC或MYSQL\_NUM，相应的mysql\_fetch\_array()调用将分别相当于mysql\_fetch\_assoc()或mysql\_fetch\_row()调用。

```
<?php
$conn_id = @mysql_connect ("cobra.snake.net", "sampadm", "secret")
    or die ("Could not connect\n");
mysql_select_db ("sampdb")
    or die ("Could not select database\n");
$query = "SELECT last_name, first_name FROM president";
$result_id = mysql_query ($query)
    or die ("Query failed\n");
while ($row = mysql_fetch_array ($result_id))
{
    # print each name twice, once using numeric indices,
```

```

        # once using associative (name) indices
        printf ("%s %s<br />\n", $row[0], $row[1]);
        printf ("%s %s<br />\n", $row["last_name"], $row["first_name"]);
    }
    mysql_free_result ($result_id);
?>

```

#### • array

**mysql\_fetch\_assoc** ( resource result\_id );

如果调用成功，这个函数将把给定结果集里的下一个数据行返回为一个关联数组；如果已经到达结果集里的最后一个数据行，则将返回false。这个数组里的元素需要通过你在用来生成这个结果集的数据库查询命令里所列举的各有关数据列的名称去访问。注意：关联键字需要区分字母的大小写情况，所以必须按与你在有关的数据库查询命令里所使用的数据列名称完全一致的形式来给出它们。

mysql\_fetch\_assoc()函数与mysql\_fetch\_array()函数在其第二个输入参数（即result\_type）被设置为MYSQL\_ASSOC时的调用效果是一样的，它们返回的结果集都不能通过数值下标去访问。

mysql\_fetch\_assoc()函数最早出现于PHP 4.0.3版本。

```

<?php
    $conn_id = @mysql_connect ("cobra.snake.net", "sampadm", "secret")
        or die ("Could not connect\n");
    mysql_select_db ("sampdb")
        or die ("Could not select database\n");
    $query = "SELECT last_name, first_name FROM president";
    $result_id = mysql_query ($query)
        or die ("Query failed\n");
    while ($row = mysql_fetch_assoc ($result_id))
    {
        printf ("%s %s<br />\n", $row["last_name"], $row["first_name"]);
    }
    mysql_free_result ($result_id);
?>

```

#### • object

**mysql\_fetch\_field** ( resource result\_id  
[, int col\_num ] );

如果调用成功，这个函数将返回指定结果集里的指定数据列的元数据信息；如果那个数据列不存在，则将返回false。如果col\_num参数缺失，连续调用mysql\_fetch\_field()函数将顺序返回结果集里各有关数据列的元数据信息；如果已经到达结果集里的最后一个数据列，则返回false。如果给出了col\_num参数，它应该落在从0到mysql\_num\_fields()-1的范围内——此时，这个函数将返回指定数据列的元数据信息；如果col\_num参数值没有落在这个范围内，则将返回false。

注意，这个函数的返回值是一个对象，有关信息将被返回为这个对象的属性，如表H-2所示。



```

<?php
$conn_id = @mysql_connect ("cobra.snake.net", "sampadm", "secret")
    or die ("Could not connect\n");
$result_id = mysql_list_fields ("sampdb", "president")
    or die ("Query failed\n");
print ("Table: sampdb.president<br />\n");
# get column metadata
for ($i = 0; $i < mysql_num_fields ($result_id); $i++)
{
    printf ("Information for column %d:<br />\n", $i);
    $meta = mysql_fetch_field ($result_id);
    if (!$meta)
    {
        print ("No information available<br />\n");
        continue;
    }
    print ("<pre>\n");
    printf ("blob:           %s\n", $meta->blob);
    printf ("def:           %s\n", $meta->def);
    printf ("max_length:      %s\n", $meta->max_length);
    printf ("multiple_key:    %s\n", $meta->multiple_key);
    printf ("name:           %s\n", $meta->name);
    printf ("not_null:        %s\n", $meta->not_null);
    printf ("numeric:         %s\n", $meta->numeric);
    printf ("primary_key:     %s\n", $meta->primary_key);
    printf ("table:          %s\n", $meta->table);
    printf ("type:           %s\n", $meta->type);
    printf ("unique_key:      %s\n", $meta->unique_key);
    printf ("unsigned:        %s\n", $meta->unsigned);
    printf ("zerofill:        %s\n", $meta->zerofill);
    print ("</pre>\n");
}
mysql_free_result ($result_id);
?>

```

表H-2 mysql\_fetch\_field()返回值的属性

属性名称	可取值及含义
blob	1——数据列是BLOB (或TEXT) 类型; 0——其他类型
def	数据列的默认值
max_length	在结果集里的数据列值的最大长度
multiple_key	1——数据列是某个非惟一化索引的组成部分; 0——其他情况
name	数据列的名称
not_null	1——数据列不允许包含NULL值; 0——允许
numeric	1——数据列是某种数值类型; 0——其他类型
primary_key	1——数据列是某个PRIMARY KEY的组成部分; 0——其他情况
table	数据列所在的数据表的名称 (如果数据列是某个表达式的计算结果, 则此属性为空)
type	数据列的类型的名称
unique_key	1——数据列是某个UNIQUE索引的组成部分; 0——其他情况
unsigned	1——数据列具备UNSIGNED属性; 0——其他情况
zerofill	1——数据列具备ZEROFILL属性; 0——其他情况

注意, `mysql_fetch_field()`所返回的对象中的`def`成员只有在结果集是通过`mysql_list_fields()`调用而获得的前提下才是有效的; 否则, 它将是一个空字符串。

`type`成员的可取值及其含义见表H-3。如果数据列是某个表达式的计算结果, 与之对应的`type`属性值将反映该表达式的类型。`type`属性的`"unknown"`取值通常表明这样一种情况: MySQL服务器比PHP所使用的MySQL客户程序开发库的版本更新, 因而MySQL服务器所知道的某些新类型还不能为MySQL客户程序开发库所识别。

表H-3 `mysql_fetch_field()`返回值的`type`属性

可取值	含 义
<code>blob</code>	BLOB (或TEXT) 数据列
<code>date</code>	DATE数据列
<code>datetime</code>	DATETIME数据列
<code>int</code>	整数型数值数据列
<code>null</code>	数据列只包含NULL值
<code>real</code>	浮点型数值数据列
<code>string</code>	字符串数据列 (但不是BLOB或TEXT类型)
<code>time</code>	TIME数据列
<code>timestamp</code>	TIMESTAMP数据列
<code>unknown</code>	数据列类型不详
<code>year</code>	YEAR数据列
<code>array</code>	<code>mysql_fetch_lengths ( resource result_id );</code>

- `array`

**`mysql_fetch_lengths (resource result_id);`**

如果调用成功, 这个函数的返回值将是一个数组, 里面是最近一次的`mysql_fetch_array()`、`mysql_fetch_assoc()`、`mysql_fetch_object()`或`mysql_fetch_row()`函数调用所返回的那个数据行的各数据列取值的长度。该数组的下标范围是从0到`mysql_num_fields()-1`。如果数据行尚未取回或者执行出错, 则将返回`false`。

```
<?php
$conn_id = @mysql_connect ("cobra.snake.net", "sampadm", "secret")
    or die ("Could not connect\n");
mysql_select_db ("sampdb")
    or die ("Could not select database\n");
$query = "SELECT * FROM president";
$result_id = mysql_query ($query)
    or die ("Query failed\n");
$row_num = 0;
while (mysql_fetch_row ($result_id))
{
    ++$row_num;
    # get lengths of column values
    printf ("Lengths of values in row %d:<br />\n", $row_num);
    $len = mysql_fetch_lengths ($result_id);
    if (!$len)
    {
```

```

        print ("No information available<br />\n");
        break;
    }
    print ("<pre>\n");
    for ($i = 0; $i < mysql_num_fields ($result_id); $i++)
        printf ("column %d: %s\n", $i, $len[$i]);
    print ("</pre>\n");
}
mysql_free_result ($result_id);
?>

```

#### • object

**mysql\_fetch\_object** ( resource result\_id  
[, int result\_type ] );

如果调用成功，这个函数将把给定结果集里的下一个数据行返回为一个对象（如果已到达结果集里的最后一个数据行，则将返回false），构成该数据行的各数据列取值将被返回为这个对象的属性，而这些属性的名字就是用来生成这个结果集的那条查询命令所选取的各数据列的名字。

这个函数的result\_type参数有MYSQL\_ASSOC（只返回关联键字）、MYSQL\_NUM（只返回数值下标）和MYSQL\_BOTH（同时返回两套下标）三种可取值。如果result\_type参数缺失，其默认值将是MYSQL\_BOTH。但有一点需要注意：因为数字不是PHP对象的合法属性名，所以如果你想通过数值下标去访问这个函数所返回的有关数据，就只能采取一些变通办法，比如把这个对象当做一个数组来对待等。

```

<?php
    $conn_id = @mysql_connect ("cobra.snake.net", "sampadm", "secret")
        or die ("Could not connect\n");
    mysql_select_db ("sampdb")
        or die ("Could not select database\n");
    $query = "SELECT last_name, first_name FROM president";
    $result_id = mysql_query ($query)
        or die ("Query failed\n");
    while ($row = mysql_fetch_object ($result_id))
        printf ("%s %s<br />\n", $row->last_name, $row->first_name);
    mysql_free_result ($result_id);
?>

```

#### • array

**mysql\_fetch\_row** ( resource result\_id );

如果调用成功，这个函数将把给定结果集里的下一个数据行返回为一个数组；如果已经到达结果集里的最后一个数据行，则将返回false。这个数组里的元素需要通过你在用来生成这个结果集的数据库查询命令里所列举的各有关数据列在结果集里的序号去访问，而这个序号必须落在从0到mysql\_num\_fields()-1的范围内。

mysql\_fetch\_row()函数与mysql\_fetch\_array()函数在其第二个输入参数（即result\_type）被设置为MYSQL\_NUM时的调用效果是一样的，它们返回的结果集都不能通过关联键字去访问。

```
<?php
    $conn_id = @mysql_connect ("cobra.snake.net", "sampadm", "secret")
        or die ("Could not connect\n");
    mysql_select_db ("sampdb")
        or die ("Could not select database\n");
    $query = "SELECT last_name, first_name FROM president";
    $result_id = mysql_query ($query)
        or die ("Query failed\n");
    while ($row = mysql_fetch_row ($result_id))
        printf ("%s %s<br />\n", $row[0], $row[1]);
    mysql_free_result ($result_id);
?>
```

#### • string

**mysql\_field\_flags** ( resource result\_id,  
int col\_num );

如果调用成功，这个函数将把给定结果集里指定数据列的元数据信息返回为一个字符串；如果执行出错，则将返回false。这个字符串由一些以空格为分隔符的单词构成，这些单词与数据列的相关属性相互对应：如果数据列具备相应的属性（即相应的标志值是true），与之对应的单词就会出现在这个字符串里；如果数据列不具备相应的属性（即相应的标志值是false），与之对应的单词就不会出现在这个字符串里。表H-4列出了可能出现在这个字符串里的各种单词及其含义。此外，col\_num参数所给出的数据列序号必须落在从0到mysql\_num\_fields()-1的范围内。

```
<?php
    $conn_id = @mysql_connect ("cobra.snake.net", "sampadm", "secret")
        or die ("Could not connect\n");
    mysql_select_db ("sampdb")
        or die ("Could not select database\n");
    $query = "SELECT * FROM member";
    $result_id = mysql_query ($query)
        or die ("Query failed\n");
    for ($i = 0; $i < mysql_num_fields ($result_id); $i++)
    {
        printf ("column %d:", $i);
        printf (" name %s,", mysql_field_name ($result_id, $i));
        printf (" flags %s\n", mysql_field_flags ($result_id, $i));
        print ("<br />\n");
    }
    mysql_free_result ($result_id);
?>
```

binary属性只有在指定数据列是一个需要区分字母大小写情况的字符串数据列（比如那些在定义时明确地使用了BINARY关键字或者被定义为BLOB类型的字符串数据列）时才会被设置。

mysql\_field\_flags()函数所返回的字符串里的各个单词可以用explode()函数分离出来，如下

所示:

```
$words = explode (" ", mysql_field_flags ($result_id, $i ) );
```

表H-4 mysql\_field\_flags()返回值里的可取值

属性名称	含 义
auto_increment	数据列具备AUTO_INCREMENT属性
binary	数据列具备BINARY属性
blob	数据列是BLOB或TEXT类型
enum	数据列是ENUM类型
multiple_key	数据列是某个非惟一化索引的组成部分
not_null	数据列不允许包含NULL值
primary_key	数据列是某个PRIMARY KEY的组成部分
timestamp	数据列是TIMESTAMP类型
unique_key	数据列是某个UNIQUE索引的组成部分
unsigned	数据列具备UNSIGNED属性
zerofill	数据列具备ZEROFILL属性

#### • int

**mysql\_field\_len** ( resource result\_id,  
int col\_num );

这个函数将返回给定结果集里指定数据列里的数据值最大长度。col\_num参数所给出的数据列序号必须落在从0到mysql\_num\_fields()-1的范围内。

```
<?php
$conn_id = @mysql_connect ("cobra.snake.net", "sampadm", "secret")
    or die ("Could not connect\n");
mysql_select_db ("sampdb")
    or die ("Could not select database\n");
$query = "SELECT * FROM member";
$result_id = mysql_query ($query)
    or die ("Query failed\n");
for ($i = 0; $i < mysql_num_fields ($result_id); $i++)
{
    printf ("column %d:", $i);
    printf (" name %s,", mysql_field_name ($result_id, $i));
    printf (" len %d\n", mysql_field_len ($result_id, $i));
    print ("<br />\n");
}
mysql_free_result ($result_id);
?>
```

#### • string

**mysql\_field\_name** ( resource result\_id,  
int col\_num );

这个函数将返回给定结果集里指定数据列的名称。col\_num参数所给出的数据列序号必须



落在从0到mysql\_num\_fields()-1的范围内。

```
<?php
    $conn_id = @mysql_connect ("cobra.snake.net", "sampadm", "secret")
        or die ("Could not connect\n");
    mysql_select_db ("sampdb")
        or die ("Could not select database\n");
    $query = "SELECT * FROM president";
    $result_id = mysql_query ($query)
        or die ("Query failed\n");
    # get column names
    for ($i = 0; $i < mysql_num_fields ($result_id); $i++)
    {
        printf ("Name of column %d: ", $i);
        $name = mysql_field_name ($result_id, $i);
        if (!$name)
            print ("No name available<br />\n");
        else
            print ("{$name}<br />\n");
    }
    mysql_free_result ($result_id);
?>
```

• int

**mysql\_field\_seek** ( resource result\_id,  
int col\_num );

每一个SELECT查询所返回的结果集里都有一个供内部使用的数据列指针，这个指针的作用是告诉mysql\_fetch\_field()函数在下一次被调用时应该返回关于哪一个数据列的元数据信息。如果你想把结果集里的这个内部指针设置为指向某个特定的数据列，就需要调用mysql\_field\_seek()函数来达到这一目的。这样，如果在发出下一个mysql\_fetch\_field()调用的时候没有给出其数据列序号参数，它将默认地返回你在mysql\_field\_seek()调用里以序号参数col\_num指定的那个数据列的元数据信息。col\_num必须落在从0到mysql\_num\_fields()-1的范围内。如果数据列序号参数合法且调用成功，这个函数将返回true；否则，将返回false。

```
<?php
    $conn_id = @mysql_connect ("cobra.snake.net", "sampadm", "secret")
        or die ("Could not connect\n");
    mysql_select_db ("sampdb")
        or die ("Could not select database\n");
    $query = "SELECT * FROM president";
    $result_id = mysql_query ($query)
        or die ("Query failed\n");
    # get column metadata
    for ($i = 0; $i < mysql_num_fields ($result_id); $i++)
    {
        printf ("Information for column %d:<br />\n", $i);
```

```

        if (!mysql_field_seek ($result_id, $i))
        {
            print ("Cannot seek to column<br />\n");
            continue;
        }
        $meta = mysql_fetch_field ($result_id, $i);
        if (!$meta)
        {
            print ("No information available<br />\n");
            continue;
        }
        print ("<pre>\n");
        printf ("blob:           %s\n", $meta->blob);
        printf ("max_length:    %s\n", $meta->max_length);
        printf ("multiple_key: %s\n", $meta->multiple_key);
        printf ("name:          %s\n", $meta->name);
        printf ("not_null:      %s\n", $meta->not_null);
        printf ("numeric:       %s\n", $meta->numeric);
        printf ("primary_key:   %s\n", $meta->primary_key);
        printf ("table:         %s\n", $meta->table);
        printf ("type:          %s\n", $meta->type);
        printf ("unique_key:    %s\n", $meta->unique_key);
        printf ("unsigned:      %s\n", $meta->unsigned);
        printf ("zerofill:      %s\n", $meta->zerofill);
        print ("</pre>\n");
    }
    mysql_free_result ($result_id);
?>

```

- string

**mysql\_field\_table** ( resource result\_id,  
                                int col\_num );

这个函数将返回给定结果集里指定数据列所在的数据表的名称。如果数据列是某个表达式的计算结果，与之对应的数据表名将是一个空字符串。col\_num参数所给出的数据列序号必须落在从0到mysql\_num\_fields()-1的范围内。

```

<?php
    $conn_id = @mysql_connect ("cobra.snake.net", "sampadm", "secret")
        or die ("Could not connect\n");
    mysql_select_db ("sampdb")
        or die ("Could not select database\n");
    $query = "SELECT * FROM president";
    $result_id = mysql_query ($query)
        or die ("Query failed\n");
    for ($i = 0; $i < mysql_num_fields ($result_id); $i++)
    {
        printf ("column %d:", $i);
    }

```

```

        printf (" name %s,", mysql_field_name ($result_id, $i));
        printf (" table %s\n", mysql_field_table ($result_id, $i));
        print ("<br />\n");
    }
    mysql_free_result ($result_id);
?>

```

#### • string

**mysql\_field\_type** ( resource result\_id,  
int col\_num );

这个函数将返回给定结果集里指定数据列的类型的名称，它应该是mysql\_fetch\_field()函数的返回值里的type属性的某一个可取值（见表H-3）。col\_num参数所给出的数据列序号必须落在从0到mysql\_num\_fields()-1的范围内。

```

<?php
    $conn_id = @mysql_connect ("cobra.snake.net", "sampadm", "secret")
        or die ("Could not connect\n");
    mysql_select_db ("sampdb")
        or die ("Could not select database\n");
    $query = "SELECT * FROM president";
    $result_id = mysql_query ($query)
        or die ("Query failed\n");
    for ($i = 0; $i < mysql_num_fields ($result_id); $i++)
    {
        printf ("column %d:", $i);
        printf (" name %s,", mysql_field_name ($result_id, $i));
        printf (" type %s\n", mysql_field_type ($result_id, $i));
        print ("<br />\n");
    }
    mysql_free_result ($result_id);
?>

```

#### • bool

**mysql\_free\_result** ( resource result\_id );

释放与指定结果集相关联的各种资源。通过mysql\_query()、mysql\_unbuffered\_query()、mysql\_list\_dbs()、mysql\_list\_fields()、mysql\_list\_processes()、mysql\_list\_tables()等函数创建（返回）的结果集都可以用这个函数来释放。

PHP会在脚本执行结束时自动释放所有尚未被释放的结果集，但如果你的脚本会生成很多结果集，就应该在适当的时候调用这个函数来释放那些已经不再需要的结果集。下面是某个PHP脚本里的一条循环语句，它在执行时将消耗大量的内存：

```

<?php
    for ($i = 0; $i < 10000; $i++)
    {
        $result_id = mysql_query ("SELECT * from president");
    }
?>

```

如果我们在这个循环语句中的mysql\_query()调用的后面加上一个mysql\_free\_result()调用,就可以把用来容纳结果集的内存量压缩到非常小的地步:

```
<?php
    for ($i = 0; $i < 10000; $i++)
    {
        $result_id = mysql_query ("SELECT * from president");
        mysql_free_result ($result_id);
    }
?>
```

此外,如果数据库查询命令是通过mysql\_unbuffered\_query()调用发出的而你有可能在到达其结果集里的最后一个数据行之前就退出相应的数据行取回和处理循环,就应该明确地发出一个mysql\_free\_result()调用来释放这个结果集——mysql\_free\_result()将自动取回并丢弃所有尚未被取回的数据行。如果遗漏了这一步骤,你发出的下一条数据库查询命令将无法顺利执行,它将导致一个“out of sync”(结果集不同步)错误。

#### • int

**mysql\_insert\_id** ([ resource conn\_id ] );

返回指定连接上最近一次发出的数据库查询命令所生成的AUTO\_INCREMENT值。但如果自该连接建立以来还没有生成过一个这样的值,则将返回0。一般说来,在发出可能会生成一个新AUTO\_INCREMENT值的数据库查询命令之后,应该马上就去调用mysql\_insert\_id()函数。如果在调用mysql\_insert\_id()函数之前又发出了另一条数据库查询命令,这个函数所返回的就可能不是刚才生成的那个编号值了。

注意,PHP函数mysql\_insert\_id()与SQL函数LAST\_INSERT\_ID()的行为是不一样的:前者由客户端负责管理,从客户端发出的每一条数据库查询命令都有可能使它的返回值发生变化;后者则由服务器端负责管理,它的返回值不会随查询命令而变化(也就是说,即使你发出了很多条会生成一个新AUTO\_INCREMENT值的数据库查询命令,也可以根据各查询命令的句柄把它与它所生成的AUTO\_INCREMENT值对应起来)。

mysql\_insert\_id()调用的返回值特定于具体的连接,发生在其他连接上的AUTO\_INCREMENT活动不会影响到这条连接上的mysql\_insert\_id()返回值。

请注意:应该尽量避免使用这个函数来获得某个BIGINT数据列上的AUTO\_INCREMENT值,因为BIGINT类型的取值范围超过了PHP用来在其内部存放mysql\_insert\_id()返回值的有关类型。虽然PHP使用手册给出了一个解决方案——用发出SQL查询SELECT LAST\_INSERT\_ID()的办法来代替PHP函数调用mysql\_insert\_id(),可这个办法并不保险,因为这个SQL查询的结果在PHP脚本中的数值上下文里会被自动转换为一个浮点数值。

```
<?php
    $conn_id = @mysql_connect ("cobra.snake.net", "sampadm", "secret")
        or die ("Could not connect\n");
    mysql_select_db ("sampdb")
        or die ("Could not select database\n");
    $query = "INSERT INTO member (last_name,first_name,expiration)"
```

```

        . " VALUES('Brown','Marcia','2002-6-3')";
$result_id = mysql_query ($query)
    or die ("Query failed\n");
printf ("membership number for new member: %d\n", mysql_insert_id());
?>

```

• int

**mysql\_num\_fields** ( resource result\_id );

返回给定结果集里的数据列个数。

```

<?php
    $conn_id = @mysql_connect ("cobra.snake.net", "sampadm", "secret")
        or die ("Could not connect\n");
    mysql_select_db ("sampdb")
        or die ("Could not select database\n");
    $query = "SELECT * FROM president";
    $result_id = mysql_query ($query)
        or die ("Query failed\n");
    printf ("Number of columns: %d\n", mysql_num_fields ($result_id));
    mysql_free_result ($result_id);
?>

```

• int

**mysql\_num\_rows** ( resource result\_id );

返回给定结果集里的数据行个数。

```

<?php
    $conn_id = @mysql_connect ("cobra.snake.net", "sampadm", "secret")
        or die ("Could not connect\n");
    mysql_select_db ("sampdb")
        or die ("Could not select database\n");
    $query = "SELECT * FROM president";
    $result_id = mysql_query ($query)
        or die ("Query failed\n");
    printf ("Number of rows: %d\n", mysql_num_rows ($result_id));
    mysql_free_result ($result_id);
?>

```

• mixed

**mysql\_result** ( resource result\_id, int row\_num  
[, mixed column ] );

给定一个结果集（由标识符result\_id指定），返回由row\_num参数指定的那个数据行里的一个值。具体返回哪个数据列要取决于column参数，它既可以是一个数值型的数据列下标，也可以是一个你在数据库查询命令里给出的数据列名字（即关联键字）。数据列的名字还允许以tbl\_name.col\_name的形式给出（tbl\_name是数据列col\_name所在的数据表的名字；如果你在数据库查询命令里使用了这个数据表的别名，PHP还允许你以alias\_name.col\_name的形式来指定这个数据列）。



这个函数的执行速度非常慢，所以应该尽量使用mysql\_fetch\_array()、mysql\_fetch\_assoc()、mysql\_fetch\_object()或mysql\_fetch\_row()等函数来代替之。如果你执意使用mysql\_result()函数，请尽量使用一个数值型数据列下标来作为column参数的值——这种做法要比把数据列的名字用做column参数值的做法速度更快。

```
<?php
    $conn_id = @mysql_connect ("cobra.snake.net", "sampadm", "secret")
        or die ("Could not connect\n");
    mysql_select_db ("sampdb")
        or die ("Could not select database\n");
    $query = "SELECT last_name, first_name FROM president";
    $result_id = mysql_query ($query)
        or die ("Query failed\n");
    for ($i = 0; $i < mysql_num_rows ($result_id); $i++)
    {
        for ($j = 0; $j < mysql_num_fields ($result_id); $j++)
        {
            if ($j > 0)
                print (" ");
            print (mysql_result ($result_id, $i, $j));
        }
        print ("<br />\n");
    }
    mysql_free_result ($result_id);
?>
```

#### • string

**mysql\_tablename** ( resource result\_id, int row\_num );

给定一个由mysql\_list\_tables()函数所返回的结果集标识符和该结果集里的一个数据行序号，如果调用成功，mysql\_tablename()函数将返回保存在该结果集指定数据行的那个数据表的名称；如果执行出错，则将返回false。数据行序号参数row\_num必须落在从0到mysql\_num\_rows()-1的范围内。

mysql\_tablename()函数实际上是mysql\_result()函数的别名。

```
<?php
    $conn_id = @mysql_connect ("cobra.snake.net", "sampadm", "secret")
        or die ("Could not connect\n");
    $result_id = mysql_list_tables ("sampdb")
        or die ("Query failed\n");
    print ("sampdb tables:<br />\n");
    for ($i = 0; $i < mysql_num_rows ($result_id); $i++)
        printf ("%s<br />\n", mysql_tablename ($result_id, $i));
    mysql_free_result ($result_id);
?>
```

## H.2.5 信息收集类函数

本小节里的函数负责返回有关客户、服务器、协议版本以及当前连接的描述信息。下面这段代码演示了各有关函数的用法：

```
<?php
    $conn_id = @mysql_connect ("cobra.snake.net", "sampadm", "secret")
        or die ("Could not connect\n");
    $info = mysql_character_set_name ()
        or die ("Could not get character set name\n");
    print ("Character set name: $info\n");
    $info = mysql_get_client_info ()
        or die ("Could not get client info\n");
    print ("Client info: $info\n");
    $info = mysql_get_host_info ()
        or die ("Could not get host info\n");
    print ("Host info: $info\n");
    $info = mysql_get_proto_info ()
        or die ("Could not get protocol info\n");
    print ("Protocol info: $info\n");
    $info = mysql_get_server_info ()
        or die ("Could not get server info\n");
    print ("Server info: $info\n");
    $info = mysql_stat ()
        or die ("Could not get status info\n");
    print ("Status info: $info\n");
    $info = mysql_thread_id ()
        or die ("Could not get thread ID\n");
    print ("Thread ID: $info\n");
?>
```

本小节里的函数大都与MySQL C API里的同名函数相对应，详细情况请参见本书附录F对那些函数的描述。

- string

**mysql\_get\_client\_info** ( void );

返回一个字符串，其内容是当前客户程序开发库的版本信息。

mysql\_get\_client\_info()函数最早出现于PHP 4.0.5版本。

- string

**mysql\_get\_host\_info** ( [ resource conn\_id ] );

返回一个字符串，其内容是对给定连接的描述信息，比如“Localhost via UNIX socket”、“cobra.snake.net via TCP/IP”或“ . via named pipe”。

mysql\_get\_host\_info()函数最早出现于PHP 4.0.5版本。

- int

**mysql\_get\_proto\_info** ( [ resource conn\_id ] );

返回一个数字，它表明了指定连接所使用的客户/服务器协议的版本。

`mysql_get_proto_info()`函数最早出现于PHP 4.0.5版本。

- string

**`mysql_get_server_info ( [ resource conn_id ] );`**

返回一个字符串，其内容是关于服务器版本的描述信息，比如"4.0.2-alpha-log"。这个版本号由版本序号和一个或者多个后缀组成。各种版本号后缀的含义可以在附录C对`VERSION()`函数的介绍内容里查到。

`mysql_get_server_info()`函数最早出现于PHP 4.0.5版本。

- string

**`mysql_info ( [ resource conn_id ] );`**

有些数据库查询命令会影响到多个数据行，这个函数所返回的就是有关这些查询命令的描述信息。这个函数与C API中的`mysql_info()`函数功能相当。

`mysql_info()`函数最早出现于PHP 4.3.0版本。

- string

**`mysql_stat ( [ resource conn_id ] );`**

返回一个字符串，里面包含着服务器的运行状态信息。这个函数与C API中的`mysql_stat()`函数功能相当。

`mysql_stat()`函数最早出现于PHP 4.3.0版本。

- int

**`mysql_thread_id ( [ resource conn_id ] );`**

返回与指定连接相关联的服务器线程标识符。这个函数与C API中的`mysql_thread_id()`函数功能相当。

`mysql_thread_id()`函数最早出现于PHP 4.3.0版本。

- string

**`mysql_character_set_name ( [ resource conn_id ] );`**

返回给定连接上的默认字符集的名称。

`mysql_character_set_name()`要求MySQL的版本不低于3.23.21。它最早出现于PHP 4.3.0版本。

## H.2.6 正逐渐被淘汰的函数

在编写PHP脚本的时候，应该尽量避免使用这一小节里的函数，因为它们正逐步被一些更新、更好的函数（见各有关条目里的说明）所替代。

- bool

**`mysql_create_db ( string db_name  
[ , resource conn_id ] );`**

以指定名称创建一个数据库。如果数据库创建成功，将返回true；如果执行出错，则将返回false。你必须有这个数据库上的CREATE权限才能执行这个操作。

建议大家采用通过`mysql_query()`调用来发出一条CREATE DATABASE语句的办法来代替使用`mysql_create_db()`函数。

- resource

```
mysql_db_query ( string db_name,
                string query
                [, resource conn_id ] );
```

这个函数与mysql\_query()调用的用途相同，但多了一个数据库名参数：在执行数据库查询命令之前，这个函数还需要先把指定数据库选为当前的默认数据库才行。

希望大家尽可能地避免使用这个函数，因为每发出一个数据库查询命令都不得不先选定一个默认数据库的做法是非常低效的。建议大家采用先发出一个mysql\_select\_db()调用来选定当前的默认数据库、再通过mysql\_query()调用来发出数据库查询命令的办法来代替使用mysql\_create\_db()函数。

- bool

```
mysql_drop_db ( string db_name
                [, resource conn_id ] );
```

丢弃（即删除）db\_name参数指定的数据库。如果数据库删除成功，将返回true；如果执行出错，则将返回false。你必须有这个数据库上的DROP权限才能执行这个操作。

请三思而后行，在执行了数据库丢弃操作之后，数据库里的数据就找不回来了。

建议大家采用通过mysql\_query()调用来发出一条DROP DATABASE语句的办法来代替使用mysql\_drop\_db()函数。

出于向后兼容的需要，PHP支持在脚本里使用一些比较老的函数名，但它们实际上都是一些新函数名的别名（见表H-5）。需要提醒大家注意的是，有些比较“新”的函数现在也处于将被淘汰的边缘。

表H-5 提供向后兼容性的函数别名

老 名 称	新 名 称
mysql_createdb ()	mysql_create_db ()
mysql_dbname ()	mysql_db_name ()
mysql_dropdb ()	mysql_drop_db ()
mysql_fieldflags ()	mysql_field_flags ()
mysql_fieldlen ()	mysql_field_len ()
mysql_fieldname ()	mysql_field_name ()
mysql_fieldtable ()	mysql_field_table ()
mysql_fieldtype ()	mysql_field_type ()
mysql_freeresult ()	mysql_free_result ()
mysql_listdbs ()	mysql_list_dbs ()
mysql_listfields ()	mysql_list_fields ()
mysql_listtables ()	mysql_list_tables ()
mysql_numfields ()	mysql_num_fields ()
mysql_numrows ()	mysql_num_rows ()
mysql_selectdb ()	mysql_select_db ()

## 附录I 挑 选 ISP

现在，很多人都拥有一条全天24小时在线的高速因特网连接和一条在必要时使用的MySQL服务器连接。这种情况在大学校园环境或者那些自己拥有计算机部门的大公司里尤其多见。但这种享受并不是人人都能得到，还有很多人必须通过因特网服务提供商（Internet Service Provider，简称ISP）才能上网。除电子邮件和Web浏览等基本的因特网服务项目外，很多ISP还提供了一些其他的服务项目，比如MySQL数据库检索服务、Web站点托管服务、允许使用Perl或PHP等语言进行程序设计等等。

既然如此，怎样才能挑选一家最能满足你要求的ISP呢？本附录将就这一问题给出一些指导意见。下面的讨论适用于两种情况：1）你不仅需要连接到因特网，还需要连接到一台能够提供MySQL服务的主机上去；2）你本人的机器上已经运行有MySQL并想让别人能够通过因特网来访问你的机器，现在需要寻找一家能够提供这种连接服务的ISP。

寻找候选ISP的办法之一是访问有关软件的官方Web站点，找找它们有没有提供一份能够提供相应服务的ISP名单。比如说，在PHP的官方Web站点上就有一个搜索页面是专门用来寻找有哪些ISP能够提供PHP服务的，这个页面里有一个搜索条件能让你把还同时提供有MySQL服务的ISP给查出来。另外有个站点上也有一份MySQL服务提供商的名单。下面就是这两个站点的网址：

```
http: //hosts.php.net/ search.php  
http: //www.wix.com/ mysql-hosting/
```

ISP提供的服务项目可以说是千差万别，在做出选择之前，对它们进行一次认真的对比是非常重要的。一定要慎重选择，差的ISP会让你的MySQL使用经历变成一场噩梦，而好的ISP则能给你巨大的帮助。挑选一家ISP并不容易，可如果不这样做，就难免会做出错误的选择并不得不再花费时间去做这种评估。

本附录给出的挑选原则是相对而言的。你也许无法找到一个能百分之百地满足你要求的ISP，但你肯定可以找到一个能比其他ISP更能满足你大多数要求的ISP。

在进行评估的时候，一定要尽可能地做到客观公正。要知道，虽说ISP并不都是些眼睛只盯着你钱包而不想好好提供服务的坏蛋，可他们也并非全是好人。重要的是你自己要做好调查分析工作，只有这样，你才能找到一家既了解它自己又能帮助你达成目标的ISP。在进行评估的时候，一定要特别警惕那些对其从业经验和技術能力避而不谈的ISP，对那些在有关问题上避重就轻的ISP也要多留一个心眼。

### 假如你是ISP

那么，你既可以把MySQL作为一个服务项目提供给你的顾客（与其他数据库系统相比，这个项目的资金成本可以说是非常之低），也可以把MySQL作为一个工具在公司内部使用（比如用它来记录顾客的资料）。如果你正在考虑安装MySQL，不妨从反面角



度来阅读本附录。也就是说，当看到“请向ISP提出以下问题”的时候，请自问一下你能否回答那些问题；如果不能回答，要问自己为什么。当看到“请询问ISP是否提供有这项服务？”的时候，请自问一下你是否提供有那项服务；如果没有提供，要问自己还需要哪些东西才能做到那一点。

## 1.1 挑选ISP的预备知识

知己知彼，百战不殆。要想找到称心如意的ISP，先得把自己的需要弄清楚。在评估ISP能否满足你的某项要求之前，你先得知道你的要求意味着怎样的服务和带宽。我们挑选ISP的不过是为了能够上网，这个目标本身并不复杂，但要想满意地达到这个目标可不容易，它包含着好几方面的问题，有些问题可能是你根本就没想过的。为什么要上网？上网去干什么？合理的代价应该是多大？

把自己的需要弄清楚之后，就可以根据自己的需要向各候选ISP提出一组相同的问题而不是向不同的ISP提出不同的问题，从而简化你在收到ISP们的回答之后进行的对比评估工作。如果你没有把自己的需要归纳为一组需要各家候选ISP们去回答的问题，就很难用一些同样的问题去咨询每一家ISP；而如果你向不同的ISP提出的是不同的问题的话，要想对它们的回答做出有意义的评估就会很困难。

作为评估工作的一个部分，可以先给一两家ISP打电话，看它们是否提供有这方面的咨询服务。如果某个ISP提供的咨询服务真的能帮助你明确你自己的要求（而不是另外一种推销手段）的话，就意味着你已经找到了一家至少在顾客服务方面做得比较好的ISP了。

在对ISP们进行筛选的时候，有件事不可不知：在对外提供MySQL数据库访问或者Web站点托管等服务项目的机构中，有很多并不提供基本的因特网接入服务。你最终可能需要选择两家ISP：一家用来把你自己接入因特网，另一家用来容纳你的数据库服务器和Web服务器。比如说，你可能需要先选择一家本地的ISP并通过它提供的电缆调制解调器、DSL、电话拨号调制解调器等接入服务连接到因特网上，然后再选择一家专精于Web托管服务的全国性ISP来建立你自己的Web站点。不同的网络服务对带宽和服务项目本身有着不同的要求，在下面的内容里，我们将对提供不同服务的ISP分别进行讨论。

### 1.1.1 带宽

从高速的专线连接到低速的拨号调制解调器，ISP支持的因特网接入方式通常有很多种，而不同的接入方式又有着不同的收费标准和速度——速度越快，收费标准也就越高。如果你有大量的数据需要从你本人的机器传输到ISP的主机去，以拨号调制解调器方式接入的速度就可能无法满足你的需要。但如果与你的MySQL主机（这台机器属于你的ISP）有关的网络活动大都是其他人在通过你的MySQL主机里的某个Web服务器而访问你的数据库里的信息的话，你本人与ISP之间的慢速连接也不是不能容忍——因为与你的数据库有关的网络流量将主要由从你的ISP到那些人之间的下传数据构成，从你本人到ISP之间的上传数据并不一定占很大的比例。

### 1.1.2 服务

就本附录所关心的问题来说，对ISP最明显的要求是它必须能够提供MySQL服务，但以下服务也必不可少或者是最好具备：

- 一个电子邮件地址：你需要利用它来与ISP的技术支持部门保持联系，更可以利用它来加入各种MySQL邮件列表。许多ISP会为一个账户提供多个电子邮件地址，如果你还有雇员或者其他合作者的话，这种情况就非常理想了。
- 一个shell账户：要想建立和维护数据库系统，你必须能够登录到你的服务器主机并在其上运行一些命令程序才行。这里所说的shell账户将使你具备这一能力。你不仅需要用到一些标准的UNIX工具程序以及mysql、mysqldump、mysqlimport等MySQL命令行客户程序，还可能需自行安装一些软件。shell账户通常要通过SSH或Telnet来使用。建议大家选择SSH，因为它的安全性要优于Telnet。
- 与MySQL有关的增值服务：与我们这里的讨论有关的增值服务项目主要有Web站点托管、允许使用Perl或PHP等程序设计语言等。因为只有具备了这些条件，你才能去编写自己的Web脚本。
- scp或FTP服务：这是为了让你能够在你的机器与ISP的机器之间传输文件。（scp将通过SSH进行数据传输，比FTP的安全性更好。）比如说，在刚开始建立数据库的时候，你可能需要把一些原始的数据文件加载到数据库里去；等数据库运转起来之后，你可能需要把数据库系统生成的一些输出下载回你自己的机器去进行处理。你也许还想让那些访问你数据库的人也能够利用FTP来下载有关的文件。
- 域名注册和虚拟主机服务：如果你还想让自己的电子邮件地址和Web站点网址出现在你自己的域名而不是你的ISP的域名之下，就需要用到这两项服务。这有助于在网上建立你本人或者公司的形象，从而获得更好的宣传效果。you@yourbiz.com形式的电子邮件地址要比you@yourbiz.some-isp.com形式的电子邮件地址更容易让人们记住，而http://www.yourbiz.com形式的URL地址也要比http://yourbiz.some-isp.com形式的URL地址更有价值。如果拥有一个属于自己的域名，即使你更换了ISP，你的网络事业也能顺利地得以延续。而如果你的名字是与某个具体的ISP联系在一起的话，更换ISP也就意味着你必须以you@yourbiz.other-isp.com和http://yourbiz.other-isp.com的面目“重新”出现在网络世界里——哪怕你能及时通知所有的人，也肯定会遭受一些无形的损失。

### 1.2 挑选ISP：基本原则

这一节列举了一些用来挑选ISP的基本原则，下一节将着重介绍与MySQL有关的ISP挑选原则。当你接触到一家新ISP的时候，应该先把以下几个问题的答案弄清楚：

- 信息是否容易获得？
- 这家ISP的收费标准是怎样的？
- 这家ISP能够提供哪些客户端访问软件？
- 这家ISP能够提供哪些前期帮助？

- 这家ISP都支持哪几种接入方式?
- 这家ISP的数据上传连接质量如何? 带宽够不够?
- 这家ISP能够提供哪些技术支持?
- 是否有配额(内存、硬盘)? 如果有, 是多大?
- 这家ISP使用的是哪一种硬件平台?
- 这家ISP是如何处理隐私和安全问题的?
- 这家ISP的信誉怎么样?
- 这家ISP已经运营了多长时间?
- 这家ISP的自我发展计划是怎样的?

接下来, 我们将对以上问题逐一进行详细的讨论:

- **信息是否容易获得?** 当你给这家ISP打咨询电话的时候, 是不是立刻就有人接听? 如果电话很难打通, 这家ISP就很可能是某个人的“业余爱好”而不能算是一家正规的企业。如果电话的另一头是一台留言机, 这家ISP会很快地与你取得联系吗? 如果你是通过电子邮件与对方进行联系的, 你会很快收到他们的答复吗?

这家ISP电话工作人员能否立刻解答你的问题? 如果不能, 他们是马上把你转接到一位专业人员那里, 还是试图让你相信你的问题无关紧要? (“噢, 这不是问题, 我们肯定可以解决——别担心。”) 这可是一个值得注意的信号: 如果他们在还没拿到你的钱时都回答不出你的问题, 在拿到你的钱以后他们又会怎样对待你呢?

这家ISP的员工是不是用你能明白的术语来与你进行交流? 如果他们使用的都是推销辞令, 那你很可能是撞上了一些急于赚钱却不懂技术的家伙。反之, 如果对方使用的都是你听不懂的技术词汇, 你又很可能是在与一些不知道如何与“普通人”对话的技术天才打交道。

去这家ISP的Web站点上看看。它的条理是否清晰? 内容是否充实? 文字是否简单易懂? 做好一个主页远比做好一项服务要容易。如果他们连简单的事情都做不好, 你又怎么能指望他们把困难的事情做好呢?

- **这家ISP的收费标准是怎样的?** 因特网服务收费标准可以说是千差万别, 因为它要取决于相当多的因素。除正常使用情况下的月收费以外, 是否还有初装费? 月收费是固定的, 还是递增的? 月收费都包括哪些项目? 是无限时上网, 还是有一个总上网小时的上限? 如果超过这个上限, 会发生什么事情? 他们是会简单地拒绝连接(这虽会给你带来一些不便, 但能帮你控制费用), 还是会在基本月收费之外再增收费用? 如果是增收费用, 那么硬盘空间是否也包括在这些收费里?

对于拨号连接, 大多数ISP都提供有一个本地号码。但如果你居住在郊区, 别忘了把ISP提供的拨号上网号码是否属于长途这件事情搞清楚。

这家ISP是否提供有800号码以方便你外出时查看电子邮件或上网? 如果你经常出差并需要在旅途中访问因特网, 选择一家全国性ISP可能要比选择一家本地的ISP更能让你获得良好的服务。如果有800号码, 它是否是免费的(要知道, 有些800号码并不免费)?

ISP的基本收费通常会包括一定数量的硬盘空间(一般是几兆字节)在内, 超过这个额度还要再收取费用。如果你预计自己的数据库会变得很大或者如果你计划使用Web站点托管

服务，就一定要把这方面的收费情况弄明白（尤其是在你的站点将有很多图像的场合）。如果某家ISP允许你在它的主机上建立一个Web站点，那它是否会因其他人来浏览你的Web页面或来下载你的文件而消耗的带宽而向你收取费用？有些ISP会设置一个配额，如果你超过这个额度，它们就会暂时关闭你的站点直到下一个时间段。另一些ISP会根据你账户名下的数据下载流量按一定比例收取费用。还有一些ISP则会把你划分为商业性顾客（对这类顾客的收费标准往往会更高）。你应该把这些问题都弄清楚。

技术支持是免费的还是收费的？如果是免费的，是否有一定的时间限制（比如只在前30天免费）？如果是收费的，那么是按月或年收费还是按次收费？如果你多交点钱，它们能否在最短的响应时间内提供专业的技术支持？

如果ISP还向你推荐了一些预付费方案，就一定要留意这家公司的历史和信誉。预付费方案经常被一些刚入行的ISP当做一种筹集资金的手段来使用，但这些ISP也是最容易倒闭的，而一旦出现这种情况，你就很难再拿回你的钱了。

ISP是否要求你签订一份长期（比如一年或更长的时间）合同？如果你对ISP的能力没有把握，还是先签一份短期（比如一两个月）的合同比较好。签一份合同并不难，但签完之后再想反悔可就没那么简单了——你能把合同尚未执行的部分拿回来吗？

有些ISP能够提供一些虽然免费或者低费用但仍可享受其技术支持的账户，但这种账户通常会对你的上网活动做一些限制。比如说，硬盘使用量方面的限制可能会让你根本无法建立起一个有实用价值的数据库。

- **这家ISP能够提供哪些客户端访问软件？**它们是否支持你所使用的系统平台？有些ISP只支持特定种类的系统平台。虽说现在已经不像以前那样容易遇到这种事情，但还是有可能会发生的。如果ISP真的是只支持某种单一的系统平台，那它很可能是Windows系统。如果你使用的是UNIX或Mac OS，这种ISP可就无能为力了。

ISP会向你提供电子邮件客户端程序和Web浏览器吗？如果会，那它们是要求你必须使用它们提供的这些东西，还是允许你根据自己的喜好再做出选择？

- **这家ISP能够提供哪些前期帮助？**它们会不会帮你完成上网设置和账户开通等工作？如果会，这项服务是否收费（尤其是在你的要求比较特殊的时候）？

- **这家ISP都支持哪几种接入方式？**如果你将使用调制解调器来拨号上网的话，那你肯定不希望总听到电话占线的忙音。你不妨向ISP了解一下它们的“顾客/调制解调器”比率，但这往往还要取决于顾客们的网络活动。

偶尔上网查查电子邮件的顾客不像必须整天连在网上的商业顾客那样依赖于ISP的调制解调器阵列，但你还是应该了解一下你听到电话忙音（尤其是在你打算上网的那段时间内）的几率有多大。

你应该在与ISP签订合同之前亲自去试一试。用一部普通电话拨叫ISP的上网号码，听听有没有调制解调器的尖叫声。应该在每天的不同时段多做几次这样的试验。你有多少次会听到占线忙音或根本就没有响应？请根据你的亲身体验对ISP的调制解调器阵列的可用性做出评估。如果你是公司用户，那你肯定希望随时都能连通因特网。如果你是个人用户，那么偶尔听到几次忙音还是可以忍受的。



如果你觉得每次上网都得拨号比较麻烦，可以考虑选用一种全天在线的接入方式，比如使用电缆或DSL调制解调器等。这些接入方式的速度要比拨号上网方式快得多。这家ISP是否支持使用这些高速连接方式？注意，电缆或DSL连接通常需要某种高速的调制解调器或路由器才能使用。这类设备是由ISP向你提供还是要由你自己去购买？如果这类设备由ISP提供，你在安装它们的时候就不会遇到很多的问题。如果在你的站点与ISP之间将有大量的数据传输活动，你甚至还可以考虑申请一条专用线路。这家ISP能帮你去申请这类专用线路吗？

如果你真的决定使用一条全天在线的连接，那你可能还想让网络上的人们都能够访问到你本人的机器；也就是说，在必要的时候，你还想建立一个属于你自己的Web站点。但这需要两个前提：一是你的机器（或站点）上有别人感兴趣的内容；二是人们能够通过网络连接上你的服务器。一般来说，ISP向顾客提供的大都是动态IP地址而不是静态IP地址，并且通常还会替你过滤掉从网络到你机器的外来连接请求。如果你想让别人能够访问到你的机器，一个静态IP地址几乎是不可或缺的条件——如果你机器的IP地址总在变化，别人就很难在网上找到你的机器。你还需要确保你的服务器使用的监听端口没有被ISP阻塞——很多ISP会以安全方面的理由替你阻塞从网络到你机器的SMTP、HTTP或FTP请求（即禁用有关的通信端口），但ISP这样做的真正动机却是为了防止顾客消耗它们的带宽。有些ISP允许顾客申请使用未加阻塞的静态IP地址，但收费可能会高一些——因为它们知道申请这类连接的顾客肯定会占用一些带宽。

- 这家ISP的数据上传连接质量如何？带宽够不够？这家ISP与因特网主干线的距离有多远？它们是直接连在因特网主干线上，还是需要再通过某个ISP才能接通因特网主干线？它们是否有为应付网络故障而预备的备用线路？

这家ISP与因特网主干线之间的“管道”有多大？网络带宽是由连接类型决定的，也就是说，ISP所能处理的数据通信量要取决于它与因特网主干线之间的连接类型。OC3、T-3或T-1等高带宽专线能支持更高速的连接。如果ISP本身与因特网主干线之间是通过一些低带宽设备连接的，数据传输速率就会受到这些瓶颈的限制。比如说，如果ISP是通过一条DSL线路来为几十位顾客的Web站点提供托管服务的，那么可供每位顾客使用的带宽也就少得可怜了。

在对带宽进行评估的时候，一定要把ISP所服务的顾客群规模也考虑进来。请向ISP询问它们每位顾客实际享有的数据带宽有多大——即你能使用的带宽在总带宽里占据多大的比例。如果你必须与很多活跃的其他用户分享一条通往因特网主干线的连接，你实际能够使用的带宽就不会有你想像的那么大。如果你把你的Web站点托管给这样的ISP，窄小的带宽就会使人们很难连通你的Web站点，久而久之，你的站点就会乏人问津。

- 这家ISP能够提供哪些技术支持？ISP的技术支持队伍是随时待命还是只在正常的上班时间内有效？你当然可以只在正常的上班时间内去求助于ISP的技术支持队伍，但万一你在下班时间内遇到了麻烦，还能不能找到人来帮助自己呢？如果是周末，会怎么样？ISP的技术支持队伍能不能提供电话和电子邮件两套联系办法？如果你无法登录，当然也就无法发出电子邮件，所以有没有电话方式的技术支持服务很重要。电子邮件方式的技术支持服



务也很重要，因为你需要把有关的技术信息或程序输出（这些信息是很难在电话里讲清楚的）通过电子邮件发送给ISP的技术人员。它们是否承诺会在规定的时间内对顾客的求助做出响应？（虽然问题不见得能在规定的时间内得到解决，但至少应该有人来及时告诉你有关人员正替你忙着呢。）

有没有在线帮助？它的内容是清晰易懂还是模糊不清？它能不能让你迅速找到你需要的信息？在线帮助有无搜索功能？

你可以像试验拨号上网号码那样去亲身体会一下ISP的技术支持服务质量——给他们打电话。在还没有在这家ISP处开立账户之前就这样做可能会让你觉得自己有些愚蠢，但你至少可以知道自己将来会不会被安排在一个长长的电话等待队列里等上半天。问问这家ISP拥有多少名技术支持人员或者它们的“顾客/技术人员”比例是多大。

- **是否有配额（内存、硬盘）？如果有，是多大？** 这家ISP在硬盘空间和处理器时间上有没有配额方面的限制？对脚本由Web服务器调用执行的时间有没有限制性的规定？为防止因个别用户占用过多的资源而影响到其他的顾客，很多ISP会对顾客的上网活动做出一些合理的限制，但你必须清楚这些限制都是什么以及都是多少。如果你即将突破某些配额的上限，会发生什么事情？ISP是会及时通知你去做出必要的调整，还是会默默地开始额外收费而你却只能在收到下一份账单时才会知道发生了这种事？

这家ISP对电子邮件的大小有限制吗？如果你经常需要随电子邮件发送一些附件，就一定要把ISP对这方面的规定弄清楚并确保自己发出的电子邮件不会超过这个限度（很多ISP会把超过规定长度的电子邮件截短到规定长度）。

- **这家ISP使用的是哪一种硬件平台？**要知道，作为个人的爱好，在老掉牙的386个人电脑上也能运行基本的因特网服务；但作为向大量用户提供服务的ISP，这样做就不合适了，因为它根本无法承载诸如MySQL之类的大计算量服务。虽说MySQL不像其他大型数据库系统那样需要消耗很多系统资源，但你肯定希望你的ISP主机有着足够的“肌肉”。这家ISP的系统能够不吃力地承载多大的负载？

这家ISP的服务器主机安放在什么地方？是不是在某人的车库或地下室里？当负载增加的时候，座落在商业区的ISP通常能更容易（也更迅速）地让电话公司对它的中继线路进行扩容。

这家ISP对设备故障有什么样的应急措施？它们是有一套完善的事故应变计划，还是简单地把你“踢”下网去直到它们修复或者更换好设备为止？它们在过去几个月里的uptime（计算机持续运转时间）记录怎么样？如果曾经当过机，其原因是什么？这家ISP是会对文件系统进行备份，还是认为文件备份工作是你的责任？如果它们会进行备份，那么有多频繁？

这家ISP有没有预定的当机时间表（用来进行定期的系统维护和文件备份工作）？在这种事情发生之前，这家ISP会不会专门来通知你一声？如果你在ISP的主机上托管了一个Web站点，那你肯定希望它能够全天24小时持续运转。

- **这家ISP是如何处理隐私和安全问题的？** 这家ISP有没有用来保护你文件隐私权的政策？它们采取了哪些措施来防止别人盗用你的账户？
- **这家ISP的信誉怎么样？** 这家ISP还有哪些其他顾客？那些顾客对ISP所提供的服务是否满

意？ISP的口碑很重要，问问你的亲朋好友，看他们有没有听说过你的候选ISP。让这家ISP给你提供一份顾客名单也是一个办法，但他们提供的名单肯定都是些对他们的服务感到满意的人。在MySQL邮件列表上发个咨询帖子也是个好主意，你肯定能收到很多人对这家ISP的评价——是好是坏就全看你是怎么想的了。

- **这家ISP已经运营了多长时间？**ISP行业里的短命公司特别多，因为这一行业很容易起步，但要想做大做强却非常困难。如果一家ISP已经“生存”了很长时间，那它通常也不会在不久的将来就宣告倒闭，你和它的结合也就应该能维持一段相当长的时间。
- **这家ISP的自我发展计划是怎样的？**这家ISP的规模有多大？人们通常不愿意与一家刚开张的小ISP公司打交道，但最大的公司并不意味着最好的服务。规模较小的公司通常会更迅速、更主动地满足顾客的要求。

这家ISP的发展速度有多快？他们现在有多少顾客？一个月以前呢？一年以前呢？你肯定希望与有发展前途的ISP打交道，但这种发展必然会在资源、带宽、技术服务等方面也给ISP带来压力。他们对这种因发展而增加的负载有什么样的应对计划？

#### 不要对ISP有不合理的期望

没有限制的访问权限、没有限制的硬盘空间使用量以及每月只需20美元的全天候技术支持都是不切合实际的期望。你是否愿意为良好的服务和技术支持支付一定的费用？既然花了钱，你当然希望能有所回报，可ISP也希望他们所提供的服务能够得到合理的报偿。

### 1.3 挑选ISP：与MySQL有关的原则

上一节列举了一些用来挑选ISP的基本原则，本节将着重介绍与MySQL及Apache、Perl、PHP等相关服务有关的ISP挑选原则。

- 这家ISP都提供哪些与MySQL有关的服务项目？
- 这家ISP安装的是哪个版本的MySQL？
- 这家ISP在升级MySQL方面有什么样的计划？
- 这家ISP的操作系统上的MySQL有没有已知的漏洞？
- 这家ISP是否允许你安装自己的软件？
- 这家ISP是否关心你数据库的隐私和安全问题？
- 这家ISP是否会帮助你起步？
- 这家ISP提供的MySQL技术支持服务怎么样？
- 这家ISP的顾客有多少在使用着MySQL？

接下来，我们将对以上问题逐一进行详细的讨论：

- **这家ISP都提供哪些与MySQL有关的服务项目？**这家ISP应该已经把MySQL以及你需要用到的各种软件都安装好了。比较常见的情况是你已经有了一家比较满意的ISP，但它目前尚未提供MySQL服务。在这种情况下，你可以先问问他们是否愿意为你安装这些软件。

你可能需要做一些公关工作才能达到你的目的，而他们通常会要求你同意他们为与MySQL有关的各种软件所提供的技术支持达不到他们为电子邮件或Web托管等服务所提供的良好程度。

去这家ISP的Web站点上看看。上面有没有关于其MySQL服务的详细介绍？有关内容是能够表明这家ISP对MySQL有着良好的技术知识还是充斥着大量的推销性文字？这个站点容易查阅吗？这个站点上是否有一个关于他们这项服务的常见问题答疑栏目以便于你自行查阅有关的信息？

- 这家ISP安装的是什么版本的MySQL？它是一个最新的版本还是一个自安装后就再也没升级过的老版本？请查阅*MySQL Reference Manual*（MySQL参考手册），看看MySQL在这家ISP所安装的版本之后又新增了哪些功能。你是否需要用到那些新增的功能？
- 这家ISP在升级MySQL方面有什么样的计划？ISP必须在现有MySQL应用程序的稳定性和已知行为与顾客对新增功能的需求之间求得一个平衡。这个问题并不像看起来那样容易解决，所以不要只从你自己的角度出发来苛求ISP。但他们至少应该让你知道他们对MySQL的升级工作是有计划的——比如在另外一台主机上正进行着相关的测试工作。如果你愿意冒点风险去成为新系统的一名试用者，就很可能与这家ISP达成一种合作的关系，因为他们也需要一名像你这样的实际用户来测试新系统的服务功能。
- 这家ISP的操作系统上的MySQL有没有已知的漏洞？请查阅*MySQL Reference Manual*（MySQL参考手册）中与安装有关的章节，看看这家ISP所安装的MySQL有没有会影响到你数据库正常使用的漏洞。这类问题通常都会有一个规避办法。
- 这家ISP是否允许你安装自己的软件？如果允许，你就能安装一些第三方软件或者用你自己编写的程序来增强你MySQL数据库的能力。
- 这家ISP是否关心你数据库的隐私和安全问题？除为保护顾客数据而提供的各种通用手段之外，这家ISP是否还有专为保护MySQL数据而采取的特殊措施？你是否能够拥有一个完全由你支配的MySQL服务器？这对你（即顾客）是很有好处的，因为谁能看到你数据库里的内容将完全控制在你自己的手里；但这会给ISP带来相当多的工作量，同时还会加重服务器主机的资源负担。Web服务器也有着类似的情况。完全由你来支配的Web服务器当然要比与别人共享的Web服务器更安全，但Web服务器主机的处理负担就更重了。与人共享的数据库服务器和共享的Web服务器收费相对要少一些，独享的服务器收费肯定要高一些。
- 这家ISP是否会帮助你起步？当然，你不能指望ISP会教你SQL语言或者教你如何编写数据库查询命令，但他们至少应该在如何顺利连接MySQL服务器方面能够向你提供必要的帮助。他们是否能够告诉你怎样才能顺利地连接上MySQL服务器？他们是否会提供一个简单的脚本供你核实你已经进入了数据库系统？
- 这家ISP提供的MySQL技术支持服务怎么样？我曾听不少人说过他们在试图运行MySQL的时候遇到了与ISP有关的麻烦，我也注意到ISP往往会把这些问题归咎于顾客。公平地讲，有些问题的原因的确是顾客与ISP双方面的，但也有一些问题很明显是因为ISP本身并不精通MySQL却又仅凭着一知半解去试图解决这些问题而越弄越糟的。因此，很有必要在事先把候选ISP的经营管理或技术支持队伍中是否有精通MySQL的人才这一问题弄清楚。要

了解他们的技术经验和水平——ISP应该具备本书第三部分中提到的各项技能。他们是能够提供一份MySQL技术支持合同（你不妨把这份合同看做是一个表明他们重视MySQL并能帮助你解决可能出现的MySQL问题的标志），还是会无奈地告诉你说“去MySQL邮件列表查查吧”？

你还可以根据ISP授予给你的资源控制权限来判断ISP的技术支持能力。如果你想拥有你MySQL服务器上的全部控制权限而ISP真的把这些权限授予了你，就表明ISP相信你有能力管理好那个服务器。此时，你应该说服ISP允许你提供一个脚本以便在系统开机的时候能够使你的MySQL服务器也自动启动运行，这样，你就不必在系统每次重启的时候都以手动方式去启动你的MySQL服务器了。

- 这家ISP的顾客有多少在使用着MySQL？如果这家ISP的顾客有很多都在使用着MySQL，那么，与只有两个顾客在使用MySQL的那些ISP们相比，这家ISP提供的MySQL技术支持通常要更好一些。

#### 警惕“心不在焉”的系统管理员

有些ISP并不具备提供MySQL服务的能力。他们之所以要安装MySQL，纯粹是为了能够在他们的服务项目清单里增加一个用来吸引顾客的卖点。这类ISP公司中的系统管理员往往并不具备完成MySQL系统管理工作的知识和能力，因而也就很少去进行这方面的管理工作。请大家提高警惕，这种“心不在焉”的系统管理员（以及这样的ISP）还是敬而远之比较好。



[ G e n e r a l   I n f o r m a t i o n ]

书名=My SQL权威指南

作者=(美)Paul DuBois著;杨涛等译

页码=931

I S B N = 9 3 1

S S 号 = 1 1 2 2 0 1 4 5

d x N u m b e r = 0 0 0 0 0 3 1 1 2 1 2 0

出版时间=2004

出版社=该引擎未能查询到

定价:98.00

试读地址=<http://book.szdnnet.org.cn/bookDetail.jsp?dxNumber=000003112120&d=B837D91C2DC36214A54FB7952DD140EB&fenlei=1817040301010303&sw=MySQL%C8%A8%CD%FE%D6%B8%C4%CF>

全文地址=<http://img9.5read.com/image/ss2jpg.dll?did=n5&pid=58612CCDAEE9204D7CD575DA7193AFD20D014DEC10B69C3D69C4AA50B57C6548C2D0240958D9D7AB5A98CABD8FC8A7C3803ED77F696883C2EFA9051F637393A9300AAA6C0543D462593E590E860488AF7050949141E80D4B076690A878CC2F5157702174B7A0B30201AEC CA50A4058E3EA7F&jid=/>